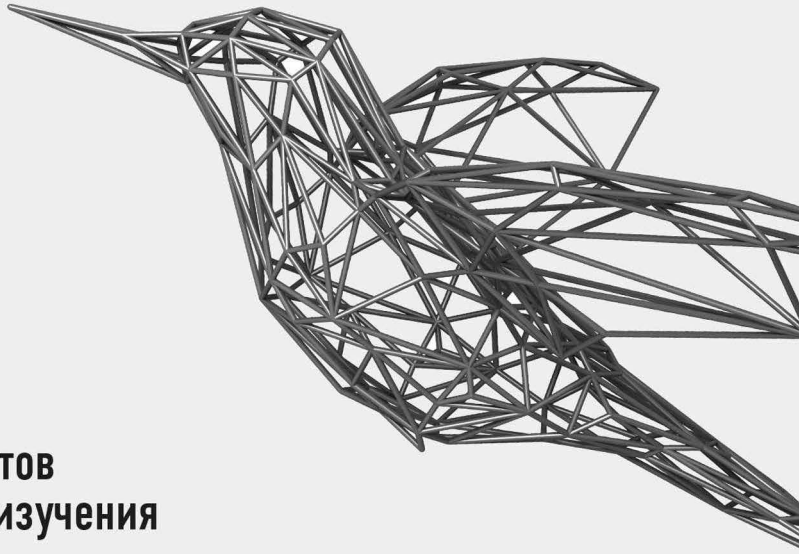
Васильев А.Н.

ПРОГРАММИРОВАНИЕ НА C# ДЛЯ НАЧИНАЮЩИХ



ОСНОВНЫЕ СВЕДЕНИЯ

- Подходит для студентов и самостоятельного изучения
- От типов данных и переменных до операторов и массивов
- Подробные примеры с разъяснениями автора
- Основные сведения о структуре языка

РОССИЙСКИЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

УДК 004.43
ББК 32.973.26-018.1
В19

Васильев, Алексей.

В19 Программирование на C# для начинающих. Основные сведения / Алексей Васильев. — Москва : Эксмо, 2018. — 592 с. — (Российский компьютерный бестселлер).

ISBN 978-5-04-092519-3

В этой книге Алексей Васильев, доктор физико-математических наук и автор популярных российских самоучителей по программированию, приглашает читателей ознакомиться с основами языка C#. Прочитав ее, вы узнаете историю языка, его структуру, ознакомитесь с типами данных и переменными, операторами, циклами и множеством другой полезной информации, необходимой для работы с этим языком.

УДК 004.43
ББК 32.973.26-018.1

ISBN 978-5-04-092519-3

ОГЛАВЛЕНИЕ

Введение
ЯЗЫК C# И ТЕХНОЛОГИЯ .NET FRAMEWORK

> Русская речь не сложнее других. Вон Маргадон — дикий человек — и то выучил.
>
> *из к/ф «Формула любви»*

Язык C# уже многие годы неизменно входит в список языков программирования, самых востребованных среди разработчиков программного обеспечения. Язык является базовым для технологии .Net Framework, разработанной и поддерживаемой корпорацией Microsoft. Именно языку C# посвящена книга, которую читатель держит в руках.

История создания языка C#

> История, леденящая кровь. Под маской овцы скрывался лев.
>
> *из к/ф «Покровские ворота»*

Язык C# создан инженерами компании Microsoft в 1998–2001 годах. Руководил группой разработчиков Андерс Хейлсберг, который до того трудился в фирме Borland над созданием компилятора для языка Pascal и участвовал в создании интегрированной среды разработки Delphi. Язык C# появился после языков программирования C++ и Java. Богатый опыт их использования был во многом учтен разработчиками C#.

> **НА ЗАМЕТКУ**
>
> Синтаксис языка C# похож на синтаксис языков C++ и Java. Но сходство внешнее. У языка C# своя уникальная концепция. Вместе с тем многие управляющие инструкции в языке C# будут знакомы тем, кто программирует в C++ и Java.

Вообще же из трех языков программирования C++, Java и C# исторически первым появился язык C++. Затем на сцену вышел язык Java. И уже после этого появился язык программирования C#.

Для понимания места и роли языка C# на современном рынке программных технологий разумно начать с языка программирования C, который в свое время стал мощным стимулом для развития программных технологий как таковых. Именно из языка C обычно выводят генеалогию языка C#. Мы тоже будем придерживаться классического подхода.

Язык программирования C появился в 1972 году, его разработал Денис Ритчи. Язык C постепенно набирал популярность и в итоге стал одним из самых востребованных языков программирования. Этому способствовал ряд обстоятельств. В первую очередь, конечно, сыграл роль лаконичный и простой синтаксис языка. Да и общая концепция языка C оказалась исключительно удачной и живучей. Поэтому когда встал вопрос о разработке нового языка, который бы поддерживал парадигму объектно ориентированного программирования (ООП), то выбор естественным образом пал на язык C: язык программирования C++, появившийся в 1983 году, представлял собой расширенную версию языка C, адаптированную для написания программ с привлечением классов, объектов и сопутствующих им технологий. В свою очередь, при создании языка программирования Java отправной точкой стал язык C++. Идеология языка Java отличается от идеологии языка C++, но при всем этом базовые управляющие инструкции и операторы в обоих языках схожи.

НА ЗАМЕТКУ

Язык программирования Java официально появился в 1995 году и стал популярным благодаря универсальности программ, написанных на этом языке. Технология, используемая в Java, позволяет писать переносимые программные коды, что исключительно важно при разработке приложений для использования в Internet.

Нет ничего удивительного, что при создании языка программирования C# традиция была сохранена: синтаксис языка C# во многих моментах будет знаком тем, кто уже программирует на C++ и Java. Хотя такое сходство — внешнее. Языки очень разные, поэтому расслабляться не стоит. Да и базовые синтаксические конструкции в языке C# имеют свои особенности. Все это мы обсудим в книге.

Особенности языка C#

> Обо мне придумано столько небылиц, что я устаю их опровергать.
>
> *из к/ф «Формула любви»*

Язык программирования C# — простой, красивый, эффективный и гибкий. С помощью программ на языке C# решают самые разные задачи. На C# можно создавать как небольшие консольные программы, так и программы с графическим интерфейсом. Код, написанный на языке C#, лаконичен и понятен (хотя здесь, конечно, многое зависит от программиста). В этом отношении язык программирования C# практически не имеет конкурентов.

НА ЗАМЕТКУ

Язык C# создавался после появления языков C++ и Java. В C# были учтены и по возможности устранены «недостатки» и «недоработки», которые есть в C++ и Java. Иногда язык C# упоминается как усовершенствованная версия языков C++ и Java, хотя концепции у них совершенно разные, так что утверждение довольно поверхностно.

Кроме собственно преимуществ языка C#, немаловажно и то, что язык поддерживается компанией Microsoft. Поэтому он идеально подходит, чтобы писать программы для выполнения под управлением операционной системы Windows.

ПОДРОБНОСТИ

Операционной системой Windows и технологией .Net Framework компании Microsoft область применения языка C# не ограничивается. Существуют альтернативные проекты (такие, например, как Mono), позволяющие выполнять программы, написанные на языке C#, под управлением операционных систем, отличных от Windows. Вместе с тем мы в книге будем исходить из того, что используются «родные» средства разработки и операционная система Windows.

Как отмечалось выше, язык C# является неотъемлемой частью технологии (или платформы) .Net Framework. Основу платформы .Net Framework составляет среда исполнения CLR (сокращение от Common Language Runtime) и библиотека классов, которая используется при программировании на языке C#.

НА ЗАМЕТКУ

Платформа .Net Framework позволяет использовать и иные языки программирования, а не только C# — например, C++ или Visual Basic. Возможности платформы .Net Framework позволяют объединять «в одно целое» программные коды, написанные на разных языках программирования. Это очень мощная технология, но для нас интерес представляет написание программных кодов на языке C#. Именно особенности и возможности языка C# мы будем обсуждать в книге.

При компилировании программного кода, написанного на языке C#, создается промежуточный код. Это промежуточный код реализован на языке MSIL (сокращение от Microsoft Intermediate Language). Промежуточный код выполняется под управлением системы CLR. Система CLR запускает JIT-компилятор (сокращение от Just In Time), который, собственно, и переводит промежуточный код в исполняемые инструкции.

ПОДРОБНОСТИ

Файл с программным кодом на языке C# сохраняется с расширением `.cs`. После компиляции программы создается файл с расширением `.exe`. Но выполнить этот файл можно только на компьютере, где установлена система .Net Framework. Такой код называется управляемым (поскольку он выполняется под управлением системы CLR).

Описанная нетривиальная схема компилирования программ с привлечением промежуточного кода служит важной цели. Дело в том, что технология .Net Framework ориентирована на совместное использование программных кодов, написанных на разных языках программирования. Базируется эта технология на том, что программные коды с разных языков программирования «переводятся» (в процессе компиляции) в промежуточный код на общем универсальном языке. Проще говоря, коды на разных языках программирования приводятся «к общему знаменателю», которым является промежуточный язык MSIL.

ПОДРОБНОСТИ

Программы, написанные на Java, тоже компилируются в промежуточный байт-код. Байт-код выполняется под управлением виртуальной машины Java. Но по сравнению с языком C# имеется принципиальное отличие. Байт-код, в который переводится при компиляции Java-программа, имеет привязку к одному языку программирования — языку

Java. И в Java схема с промежуточным кодом нужна для обеспечения универсальности программ, поскольку промежуточный байт-код не зависит от типа операционной системы (и поэтому переносим). Особенность операционной системы учитывается той виртуальной машиной Java, которая установлена на компьютере и выполняет промежуточный байт-код. Промежуточный код, который используется в технологии .Net Framework, не привязан к конкретному языку программирования. Например, что при компиляции программы на языке C#, что при компиляции программы на языке Visual Basic получаются наборы инструкций на одном и том же промежуточном языке MSIL. И нужно это, еще раз подчеркнем, для обеспечения совместимости разных программных кодов, реализованных на разных языках.

Весь этот процесс для нас важен исключительно с познавательной точки зрения. Более принципиальными являются вопросы, касающиеся особенностей собственно языка C#.

Первое, что стоит отметить: язык C# полностью объектно ориентированный. Это означает, что даже самая маленькая программа на языке C# должна содержать описание хотя бы одного класса.

НА ЗАМЕТКУ

Здесь есть методологическая проблема. Состоит она в том, что неподготовленному читателю сложно воспринимать концепцию ООП сразу, без предварительной подготовки. Мы найдем выход в том, чтобы использовать определенный шаблон при написании программ. Затем, по мере знакомства с языком C#, многие моменты станут простыми и понятными.

Как отмечалось выше, базовые синтаксические конструкции языка C# напоминают (или просто совпадают) соответствующие конструкции в языках C++ и/или Java. Но, кроме них, язык C# содержит множество интересных и полезных особенностей, с которыми мы познакомимся в книге.

ПОДРОБНОСТИ

Знакомым с языками программирования C++ и/или Java будет полезно узнать, что в языке C#, как и в языке C++, используются пространства имен, указатели, существует переопределение операторов. Также в C#, как и в Java, имеются интерфейсы, объекты реализуются через ссылки, используется система автоматической

«уборки мусора». А еще в C# используются делегаты, концепция которых идеологически близка к концепции указателей на функции в C++. Массивы в C# больше напоминают массивы Java, но вообще в C# они достаточно специфичные. Индексаторы в C# позволяют индексировать объекты — подобный механизм, основанный на переопределении оператора «квадратные скобки», есть в C++. Свойства, которые используются в C#, представляют собой нечто среднее между полем и методом, хотя, конечно, больше напоминают поле с особым режимом доступа.

Есть в языке C# и «классические» механизмы, характерные для большинства языков, поддерживающих парадигму ООП. Мы познакомимся с тем, как на основе классов и объектов в языке C# реализуется инкапсуляция, узнаем, что такое наследование и как в C# используется полиморфизм. Будет много других тем и вопросов, которые мы изучим. Например, мы узнаем (во второй части книги), как в C# создаются приложения с графическим интерфейсом, и познакомимся с основными подходами, используемыми при создании многопоточных приложений. Но обо всем этом — по порядку, тема за темой.

Программное обеспечение

Показывай свою гравицану. Если фирменная вещь — возьмем!

из к/ф «Кин-дза-дза»

Для работы с языком программирования C# на компьютере должна быть установлена платформа .Net Framework. Компилятор языка C# входит в стандартный набор этой платформы. Поэтому, если на компьютере установлена платформа .Net Framework, этого достаточно для начала программирования в C#. Обычно дополнительно еще устанавливают среду разработки, благодаря которой процесс программирования на C# становится простым и приятным.

Если мы хотим написать программу (на языке C#), сначала нам нужно набрать соответствующий программный код. Теоретически сделать это можно в обычном текстовом редакторе. В таком случае набираем в текстовом редакторе код программы и сохраняем файл с расширением `.cs` (расширение для файлов с программами, написанными на языке C#).

После того как код программы набран и файл с программой сохранен, ее следует откомпилировать. Для этого используют программу-компилятор `csc.exe`, которая устанавливается, как отмечено выше, при установке платформы .Net Framework.

ПОДРОБНОСТИ

Алгоритм действий такой: в командной строке указывается название программы-компилятора `csc.exe`, а затем через пробел указывается название файла с программой на языке C#. Допустим, мы записали код программы в файл `MyProgram.cs`. Тогда для компиляции программы в командной строке используем инструкцию `csc.exe MyProgram.cs` или `csc MyProgram.cs` (расширение `exe`-файла можно не указывать). Если компиляция проходит нормально, то в результате получаем файл с расширением `.exe`, а название файла совпадает с названием исходного файла с программой (в нашем случае это `MyProgram.exe`). Полученный в результате компиляции `exe`-файл запускают на выполнение.

Файл `csc.exe` по умолчанию находится в каталоге `C:\Windows\Microsoft.NET\Framework` внутри папки с номером версии — например, `v3.5` или `v4.0`. Также для компилирования программы из командной строки придется, скорее всего, выполнить некоторые дополнительные настройки — например, в переменных среды задать путь для поиска компилятора `csc.exe`.

Хотя такая консольная компиляция вполне рабочая, прибегают к ней редко. Причина в том, что не очень удобно набирать код в текстовом редакторе, затем компилировать программу через командную строку и запускать вручную исполнительный файл. Намного проще воспользоваться специальной программой — *интегрированной средой разработки* (IDE от Integrated Development Environment). Среда разработки содержит в себе все наиболее важные «ингредиенты», необходимые для «приготовления» такого «блюда», как программа на языке C#. При работе со средой разработки пользователь получает в одном комплекте редактор кодов, средства отладки, компилятор и ряд других эффективных утилит, значительно облегчающих работу с программными кодами. Мы тоже прибегнем к помощи среды разработки. Нам понадобится приложение Microsoft Visual Studio.

Загрузить установочные файлы можно с сайта компании Microsoft `www.microsoft.com` (на сайте следует найти страницу загрузок).

После установки среды разработки мы получаем все, что нужно для успешной разработки приложений на языке C#.

ПОДРОБНОСТИ

Приложение Microsoft Visual Studio является коммерческим. Однако у него есть некоммерческая «упрощенная» версия Visual Studio Express, которая вполне подойдет для изучения языка программирования C#.

НА ЗАМЕТКУ

Кроме языка программирования C# среда разработки Visual Studio позволяет создавать программы на языках Visual Basic и C++. Часть настроек, влияющих на функциональность среды разработки, определяется в процессе установки.

Как выглядит окно приложения Visual Studio Express (версия 2015), показано на рис. В.1.

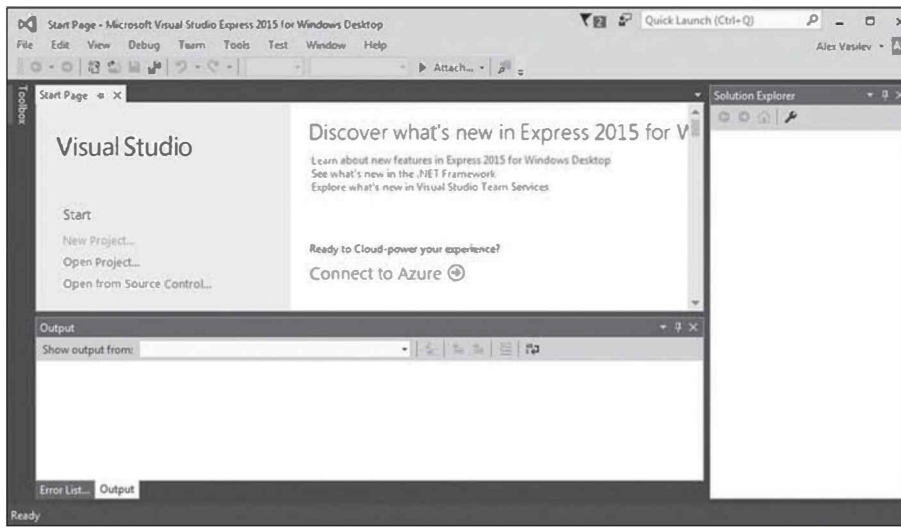


Рис. В.1. Окно приложения Visual Studio Express 2015

Основные операции, выполняемые при создании приложений с помощью среды разработки Visual Studio Express, мы кратко рассмотрим в основной части книги, когда будем обсуждать программные коды.

Среда разработки Visual Studio хотя и предпочтительная, но далеко не единственная. Существуют другие приложения, используемые при создании программ на языке C#. Причем масштабы использования альтернативных средств разработки постоянно расширяются. Но, в любом случае, имеет смысл знать, какие есть варианты помимо Visual Studio. Нас интересуют в первую очередь среды разработки для программирования на C#, распространяемые на некоммерческой основе.

Среда разработки SharpDevelop является приемлемым выбором и позволяет создавать приложения разного уровня сложности. По сути, данная среда разработки является интегрированным отладчиком, взаимодействующим со средой .Net Framework. Окно среды разработки SharpDevelop показано на рис. B.2.

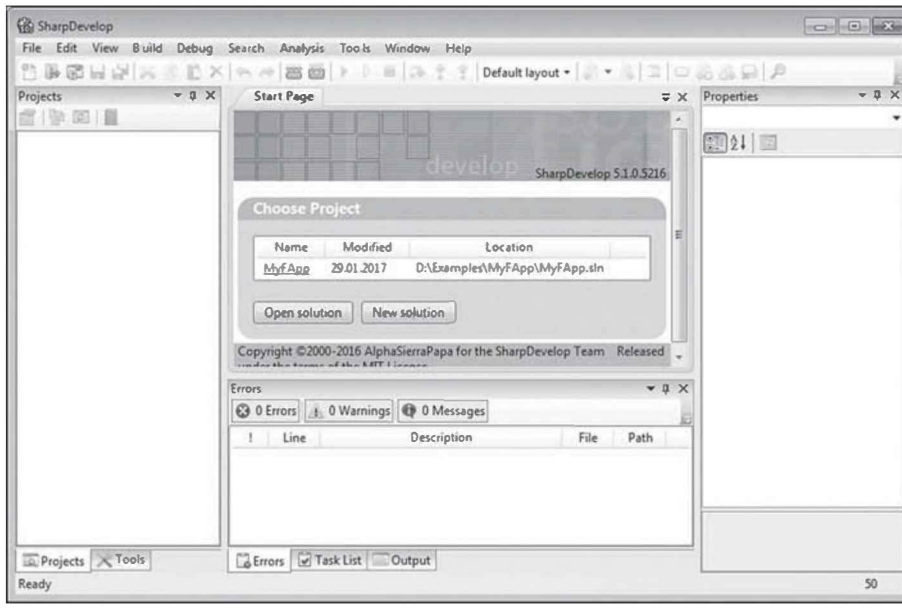


Рис. B.2. Окно среды разработки SharpDevelop

Загрузить все необходимые файлы для установки среды разработки SharpDevelop можно на сайте проекта `www.icsharpcode.net`.

Еще один проект, заслуживающий внимания, называется Mono (сайт поддержки проекта `www.mono-project.com`). Проект развивается в основном благодаря поддержке компании Xamarin (сайт `www.xamarin.com`). В рамках этого проекта была создана среда

разработки MonoDevelop (сайт `www.monodevelop.com`), предназначенная, кроме прочего, для создания приложений на языке C#. На данный момент разработчикам для установки предлагается среда разработки Xamarin Studio, которая позиционируется как продолжение проекта MonoDevelop. Окно среды разработки Xamarin Studio показано на рис. В.3.

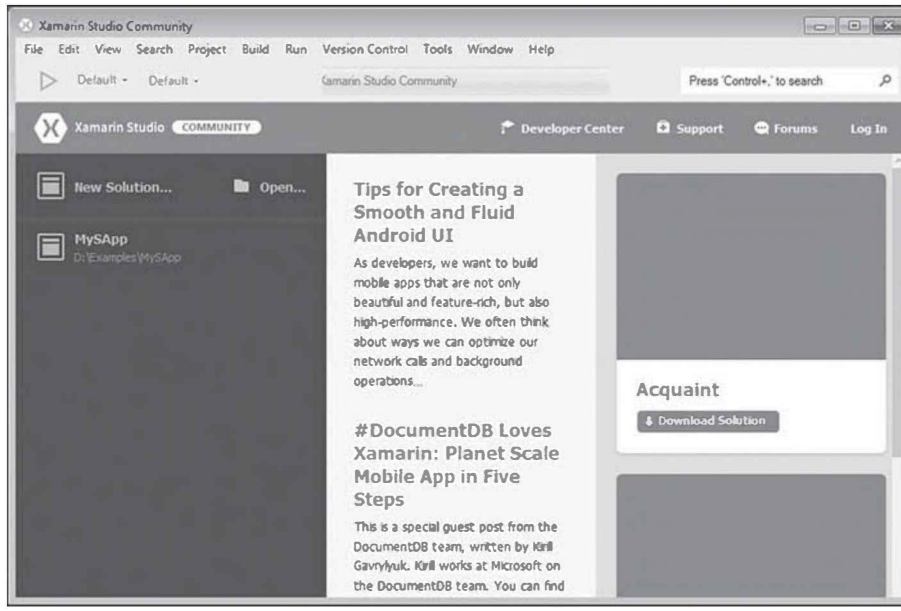


Рис. В.3. Окно среды разработки Xamarin Studio

Методы работы с перечисленными выше средами разработки не являются предметом книги. Отметим лишь следующие обстоятельства:

- У читателя есть альтернатива в плане использования среды разработки.
- Среды разработки не являются эквивалентными по своим функциональным возможностям. Наиболее надежный вариант связан с использованием среды разработки Microsoft Visual Studio. Все примеры из книги тестировались именно в этой среде разработки.
- Нужно иметь в виду, что в силу специфики сред разработки SharpDevelop и Xamarin Studio некоторые программные коды, выполняемые в Microsoft Visual Studio, могут не выполняться в SharpDevelop и Xamarin Studio (и наоборот).

- Среды разработки Microsoft Visual Studio и SharpDevelop ориентированы на использование операционной системы Windows. Среда разработки Xamarin Studio позволяет работать с операционными системами Windows, Linux и Mac OS X.

Собственно о книге

У меня есть мысль, и я ее думаю.

из м/ф «38 попугаев»

Книга предназначена для тех, кто изучает язык C# (самостоятельно, в университете или на курсах по программированию). Она содержит необходимый теоретический материал и примеры программ на языке C#. Книга состоит из двух частей. Предполагается, что каждая из частей книги будет издаваться отдельным томом.

В первой части книги рассматриваются все основные вопросы, важные при изучении языка программирования C#. Приведенный далее список не является исчерпывающим и содержит лишь названия основных тем, рассмотренных в первой части книги:

- Принципы создания программ на языке C#: структура программы, ее основные блоки, компилирование и запуск на выполнение.
- Ввод и вывод данных через консоль и через диалоговые окна.
- Переменные, базовые типы данных и основные операторы.
- Управляющие инструкции: условный оператор, оператор выбора и операторы цикла, инструкция безусловного перехода.
- Массивы и операции с ними.
- Создание классов и объектов. Основные принципы ООП.
- Перегрузка методов, конструкторы и деструкторы.
- Работа с текстом.
- Перегрузка операторов.
- Свойства и индексаторы.
- Наследование, замещение членов класса и переопределение виртуальных методов.

Освоив материал первой части, читатель сможет создавать эффективные программы высокого уровня сложности.

Во второй части книги рассматриваются такие темы:

- Абстрактные классы и интерфейсы.
- Делегаты, ссылки на методы, анонимные методы, события и лямбда-выражения.
- Перечисления и структуры.
- Указатели и операции с памятью.
- Обработка исключительных ситуаций.
- Многопоточное программирование.
- Обобщенные классы и методы.
- Создание приложений с графическим интерфейсом.
- Работа с файлами.

Темы, о которых пойдет речь во второй части книги, расширяют представление о возможностях языка C#. Есть и другие вопросы, которым уделяется внимание на страницах книги. После успешного изучения материала книги читатель приобретет необходимые знания и навыки для самостоятельного создания программ на языке C# любого уровня сложности. Понятно, что освоение тонкостей программирования на C# на этом не должно заканчиваться. Вместе с тем есть все основания полагать, что книга поможет заложить надежный фундамент, полезный в дальнейшей профессиональной карьере программиста.

Материалы, вошедшие в книгу, многократно апробировались при чтении лекций по программированию для студентов Киевского национального университета имени Тараса Шевченко и Национального технического университета Украины «Киевский политехнический институт», а также на занятиях для слушателей курсов по программированию. Поскольку при изучении программирования первоочередное значение имеют навыки, которые получают студенты и слушатели в процессе обучения, то при подборе примеров и формировании тем упор делался на прикладных аспектах программирования на языке C#. И помните, что неотъемным условием успешного изучения любого языка программирования является постоянное совершенствование практических навыков. Чтобы научиться программировать, следует постоянно тренироваться.

Это может быть как набор и отладка программных кодов, так и самостоятельное написание пусть и несложных программ. Поэтому в конце каждой главы приводится небольшой список программ для самостоятельного написания. Для лучшего закрепления материала каждая глава в книге заканчивается кратким *Резюме* с перечислением основных позиций, изложенных в соответствующей главе.

Обратная связь

— Готовы ли вы ответствовать?
— Вопрошайте.
— Готовы ли вы сказать нам всю нравду?
— Ну, всю — не всю... А что вас интересует?

из к/ф «Формула любви»

Свои замечания и комментарии по поводу этой и других книг автора можно отправить по адресу электронной почты `alex@vasilev.kiev.ua`. Также на сайте `www.vasilev.kiev.ua` можно найти краткую информацию по книге. Там же в случае необходимости размещаются дополнительные материалы к книгам.

Общение с читателями — важный этап, который позволяет понять, что в книге удалось, а что — нет. Поэтому любые критические замечания или предложения важны и востребованы. На все письма ответить, к сожалению, невозможно, но все они читаются.

Благодарности

Сильвунле, дорогие гости! Сильвунле... Жевупри авек нлезир. Господи нрости, от страха все слова новыскакивали!

из к/ф «Формула любви»

Хочется искренне поблагодарить своих студентов в университете и слушателей курсов, на которых проверялись представленные в книге примеры и отрабатывались методики. Опыт, приобретенный в общении с пытливой и требовательной аудиторией, бесценен. Надеюсь, его удалось хотя бы частично воплотить в книге.

От написания книги до выхода ее «в свет» лежит огромный путь и выполняется большой объем работ. Все это ложится на плечи сотрудников издательства, выпускающего книгу. Все эти люди заслужили огромную благодарность за их важный и благородный труд.

Сердечное спасибо моим родителям Тамаре Васильевне и Николаю Анатольевичу, детям Насте и Богдану, жене Илоне за поддержку и долготерпение, которое они проявляли все время, пока писалась эта книга.

Об авторе

— Где отличник?
— Гуляет отличник.

из к/ф «Кин-дза-дза»

Автор книги — *Васильев Алексей Николаевич*, доктор физико-математических наук, профессор кафедры теоретической физики физического факультета Киевского национального университета имени Тараса Шевченко. Сфера научных интересов: физика жидкостей и жидких кристаллов, синергетика, биофизика, математические методы в экономике, математическая лингвистика. Автор книг по математическим пакетам Maple, Mathcad, Matlab и Mathematica, офисному приложению Excel, книг по программированию на языках Java, C, C++, C#, Python и JavaScript.

Глава 1
ЗНАКОМСТВО С ЯЗЫКОМ C#

> Товарищ, там человек говорит, что он — инопланетянин. Надо что-то делать...
>
> *из к/ф «Кин-дза-дза»*

Мы начинаем знакомство с языком программирования C#. Сначала рассмотрим общие подходы:

- определимся со структурой программы;
- познакомимся с базовыми синтаксическими конструкциями, используемыми при написании программы на языке C#;
- узнаем, как набирается программный код, как программа компилируется и запускается на выполнение;
- создадим несколько несложных программ;
- познакомимся с переменными;
- узнаем, как выводятся сообщения в консоль и в специальное диалоговое окно;
- увидим, как в программу могут вводиться данные через консоль и с помощью специального диалогового окна с полем ввода.

Начнем с того, что обсудим общие принципы создания программ на языке C#.

Структура программы

> Сердце подвластно разуму. Чувства подвластны сердцу. Разум подвластен чувствам. Круг замкнулся.
>
> *из к/ф «Формула любви»*

Язык C# является полностью объектно ориентированным. Поэтому для создания даже самой маленькой программы на языке C# необходимо описать по крайней мере один *класс*. С классами (и объектами) мы познакомимся немного позже. Концепция классов и объектов является краеугольным камнем в здании ООП. Ее изучение требует определенных навыков в программировании. Здесь мы сталкиваемся с методологической проблемой: при изучении языка C# необходимо сразу использовать классы, а для объяснения того, что такое класс, желательно иметь некоторый опыт в программировании. Получается замкнутый круг. Мы его разорвем очень просто: сначала, пока только знакомимся с азами языка C#, будем использовать некоторые синтаксические конструкции языка C#, не особо вдаваясь в их смысл. В данном конкретном случае мы познакомимся с тем, как описывается класс, а что такое класс и как он используется в общем случае, обсудим немного позже.

Итак, для создания программы в C# необходимо описать класс. Описание класса начинается с ключевого слова `class`. Далее следует имя класса.

> **НА ЗАМЕТКУ**
>
> Имя класса выбираем сами. Это одно слово. Название нашего единственного класса, без особого ущерба для понимания ситуации, можем отождествлять с названием программы.

Тело класса заключается в фигурные скобки — после имени класса указываем открывающую скобку `{`, а окончание описания класса обозначается закрывающей скобкой `}`. Таким образом, шаблон описания класса имеет следующий вид (жирным шрифтом выделены обязательные элементы шаблона):

```
class имя_класса{
   // Описание класса
}
```

Возникает естественный вопрос о том, что же описывать в теле класса. В нашем случае класс будет содержать описание *главного метода*. Главный метод — это и есть программа. Выполнение программы в C# означает выполнение главного метода. Команды, которые содержит главный метод — это как раз те команды, которые выполняются при запуске программы.

НА ЗАМЕТКУ

Вообще метод (любой, не обязательно главный) — это именованный блок из команд. Для выполнения данных команд метод должен быть вызван. В C# методы описываются в классах. Есть два типа методов: статические и обычные (не статические). Для вызова обычных (не статических) методов необходимо создавать объект. Статические методы можно вызывать без создания объекта.
Главный метод является особым. Он вызывается автоматически при запуске программы на выполнение. В программе может быть один и только один главный метод.

Существует несколько альтернативных способов описания главного метода. Мы будем использовать наиболее простой шаблон, представленный ниже (жирным шрифтом выделены обязательные элементы шаблона):

```
static void Main(){
   // Код программы
}
```

Описание главного метода начинается с ключевого слова `static`. Это ключевое слово означает, что главный метод является статическим, и поэтому для вызова метода не нужно создавать объект. Далее указывается ключевое слово `void`, означающее, что метод не возвращает результат. Не возвращающий результат метод — это просто набор инструкций, которые последовательно выполняются при вызове метода. Называется главный метод `Main` (только так и никак иначе). После имени метода указываются пустые круглые скобки — пустые, потому что у главного метода нет аргументов.

ПОДРОБНОСТИ

В общем случае метод может возвращать результат. Возвращаемое методом значение подставляется вместо инструкции вызова метода после того, как метод завершает выполнение. Также у метода

могут быть аргументы — значения, которые передаются методу при вызове. Эти значения метод использует при выполнении своих внутренних инструкций. Мы используем шаблон описания главного метода, в котором главный метод не возвращает результат и аргументов у главного метода нет. Но это не единственный способ описания главного метода. Он может быть описан как такой, что возвращает результат, и аргументы ему тоже могут передаваться (аргументы командной строки). Разные способы описания главного метода обсуждаются в одной из следующих глав книги.

Также сразу отметим, что нередко кроме ключевых слов `static` и `void` в описании главного метода используется ключевое слово `public`. Ключевое слово `public` означает, что метод является открытым и доступен за пределами класса. В этом смысле его использование при описании главного метода представляется вполне логичным. Вместе с тем стандарт языка C# не предусматривает обязательного использования ключевого слова `public` в описании главного метода программы.

Тело главного метода выделяется с помощью пары фигурных скобок `{` и `}`. Напомним, что все это (то есть описание главного метода) размещается внутри тела класса. Таким образом, шаблон для создания наиболее простой программы в C# выглядит следующим образом:

```
class имя_класса{
   static void Main(){
      // Код программы
   }
}
```

В теле главного метода размещаются команды, которые должны быть выполнены в процессе работы программы. Команды выполняются одна за другой, строго в той последовательности, как они указаны в теле главного метода.

Первая программа

Для того ли я оставил свет, убежал из столицы,
чтоб погрязнуть в болоте житейском!

из к/ф «Формула любви»

Наша самая первая программа будет отображать в консольном окне сообщение. Программный код первой программы представлен в листинге 1.1.

Листинг 1.1. Первая программа

```
// Главный класс программы:
class HelloWorld{
   // Главный метод программы:
   static void Main(){
      // Отображение сообщения в консольном окне:
      System.Console.WriteLine("Изучаем язык C#");
   }
}
```

Если мы введем этот программный код в окне редактора кодов, скомпилируем и запустим программу (как это делается — описывается далее), то в консольном окне появится сообщение, представленное ниже.

Результат выполнения программы (из листинга 1.1)

```
Изучаем язык C#
```

Мы рассмотрим, как программу реализовать в среде разработки, но перед этим проанализируем программный код. Код в листинге 1.1 содержит описание класса с названием `HelloWorld`. В классе описан главный метод `Main()`. В главном методе всего одна команда `System.Console.WriteLine("Изучаем язык C#")`, которой в консольном окне отображается сообщение.

НА ЗАМЕТКУ

Все команды в C# заканчиваются точкой с запятой. Размещать команды в разных строках не обязательно, но так программный код лучше читается. Также рекомендуется выделять блоки команд

отступами. Пробелы между командами игнорируются, поэтому наличие или отсутствие отступов на выполнении программного кода не сказывается, зато наличие отступов значительно улучшает читабельность кода.

Команда означает вызов метода `WriteLine()`. Метод предназначен для отображения текста. Текст (в двойных кавычках) передается аргументом методу. Но метод не существует сам по себе. Метод вызывается из класса, который называется `Console`. Имя метода отделяется от имени класса точкой. Класс `Console` находится в пространстве имен `System`. Этот факт отображается в команде: имя класса `Console` указывается вместе с названием пространства имен `System`, в котором содержится класс, а имя класса и пространства имен разделяется точкой.

ПОДРОБНОСТИ

Метод представляет собой именованный блок команд. Другими словами, имеется некоторый набор команд, у этого блока есть имя (имя метода). При вызове метода команды из этого метода выполняются. Методу могут передаваться аргументы (значения, используемые при выполнении команд метода).

Методы описываются в классах и бывают статическими и нестатическими. Нестатический метод вызывается из объекта. Статический метод вызывается без создания объекта. При вызове статического метода указывается имя класса, в котором метод описан. В нашем случае вызывается статический метод `WriteLine()`, который описан в классе `Console`.

Классы, в свою очередь, разбиты по группам. Каждая такая группа называется пространством имен. У каждого пространства имен есть имя. Если пространство имен не включено в программу специальной инструкцией (как это делается — объясняется позже), то имя пространства имен указывается перед именем класса. Класс `Console` находится в пространстве имен `System`. Поэтому команда вызова статического метода `WriteLine()`, описанного в классе `Console` из пространства имен `System`, выглядит как `System.Console.WriteLine()`.

Программный код содержит еще и комментарии. Это пояснения, которые предназначены для человека и игнорируются при компилировании программы. Комментарий начинается с двойной косой черты `//`. Все, что находится справа от двойной косой черты `//`, при компиляции во внимание не принимается.

НА ЗАМЕТКУ

Комментарии, которые начинаются с двойной косой черты, называются однострочными, поскольку должны размещаться в одной строке. Кроме однострочных комментариев, существуют и многострочные комментарии. Такой комментарий может занимать несколько строк. Начинается многострочный комментарий с символов `/*`, а заканчивается символами `*/`.

После того как мы разобрали программный код, выясним теперь, что с этим кодом делать для того, чтобы программу можно было откомпилировать и запустить на выполнение.

Использование среды разработки

> Фимка, ну что ж ты стоишь! Неси бланманже с киселем!
>
> *из к/ф «Формула любви»*

Далее мы рассмотрим вопрос о том, как с помощью среды разработки Microsoft Visual Studio Express создать программу. Итак, первое, что нам необходимо сделать — запустить среду разработки. В окне, которое откроется, необходимо создать новый проект. Для этого в меню **File** выбираем команду **New Project**, как это показано на рис. 1.1.

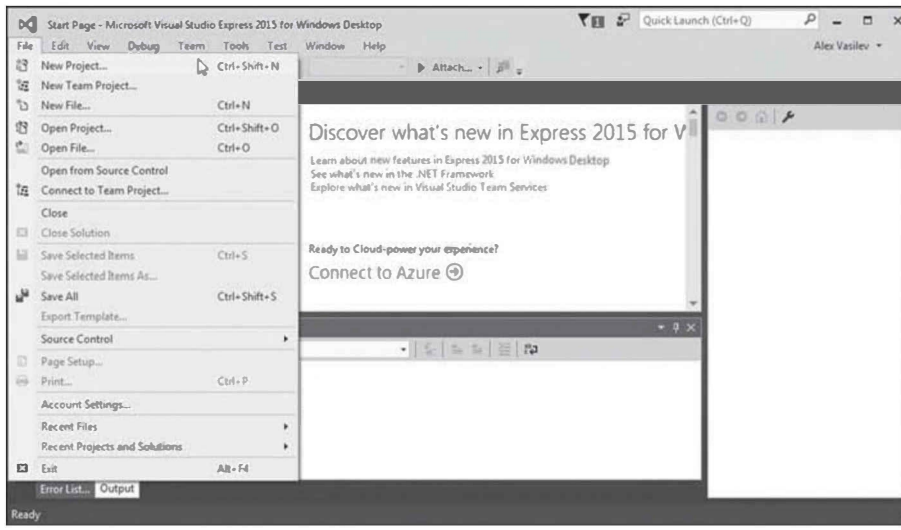


Рис. 1.1. Создание нового проекта в окне среды разработки Microsoft Visual Studio Express

НА ЗАМЕТКУ

Для создания нового проекта можно воспользоваться пиктограммой на панели инструментов или нажать комбинацию клавиш <Ctrl>+<Shift>+<N>.

В результате появляется окно создания нового проекта, показанное на рис. 1.2.

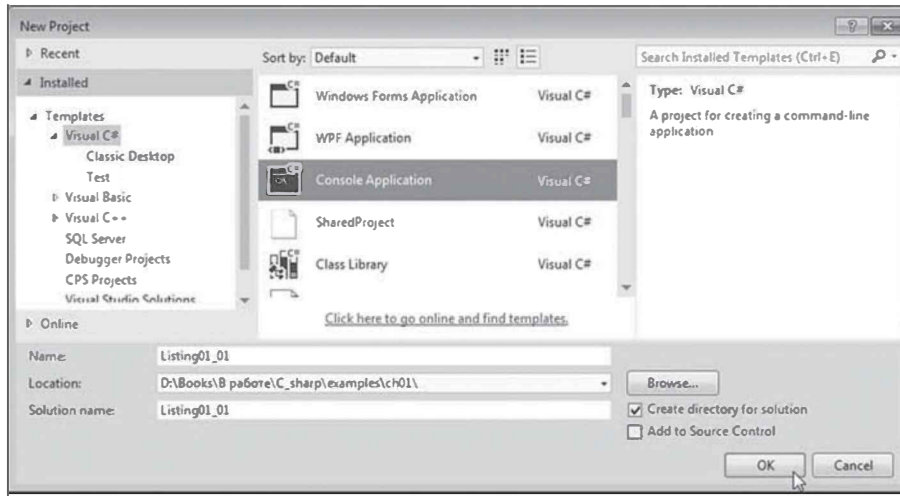


Рис. 1.2. Окно создания нового проекта New Project

В разделе **Templates** выбираем **Visual C#**, а в центральной части окна следует выбрать пункт **Console Application**. В поле **Name** указываем название проекта (мы используем название **Listing01_01**), а в поле **Location** выбираем место для сохранения файлов проектов.

НА ЗАМЕТКУ

Для выбора папки для сохранения проекта можно воспользоваться кнопкой **Browse** справа от поля **Location**.

После щелчка на кнопке **OK** открывается окно проекта с шаблонным кодом, представленное на рис. 1.3.

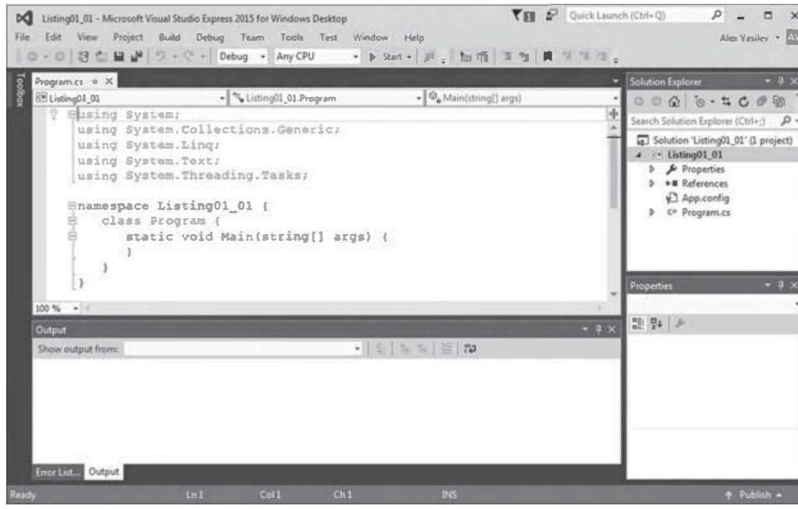


Рис. 1.3. Окно проекта с шаблонным кодом

ПОДРОБНОСТИ

При создании нового проекта программа содержит начальный шаблонный код. Он содержит наиболее часто используемые инструкции и описание класса с главным методом программы (с пустым телом). Назначение шаблонного кода — облегчить программисту работу. Среда разработки Visual Studio позволяет настраивать (задавать) шаблонный код, который автоматически добавляется в проект.

В окне редактора (окно с программным кодом) заменяем шаблонный код на код нашей программы (см. листинг 1.1). Результат показан на рис. 1.4.

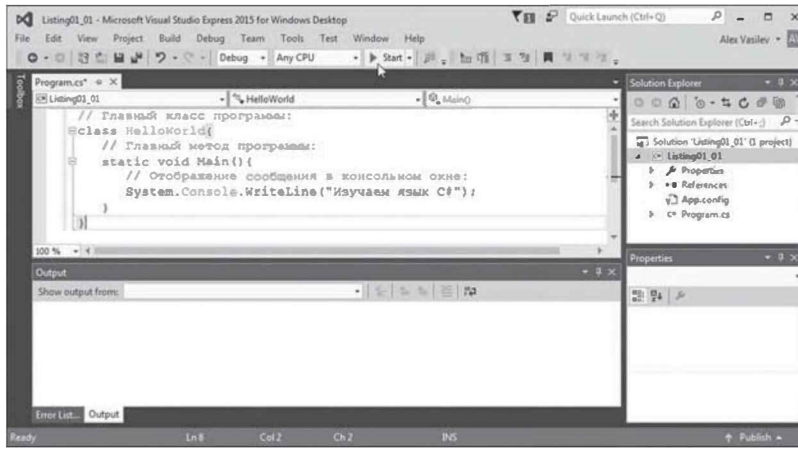


Рис. 1.4. Окно проекта с кодом программы

После того как код введен, программу можно компилировать и запускать на выполнение. Для этого можем воспользоваться специальной пиктограммой с зеленой стрелкой на панели инструментов (см. рис. 1.4). Но лучше воспользоваться командой **Start Without Debugging** из меню **Debug** (комбинация клавиш <Ctrl>+<F5>), как показано на рис. 1.5.

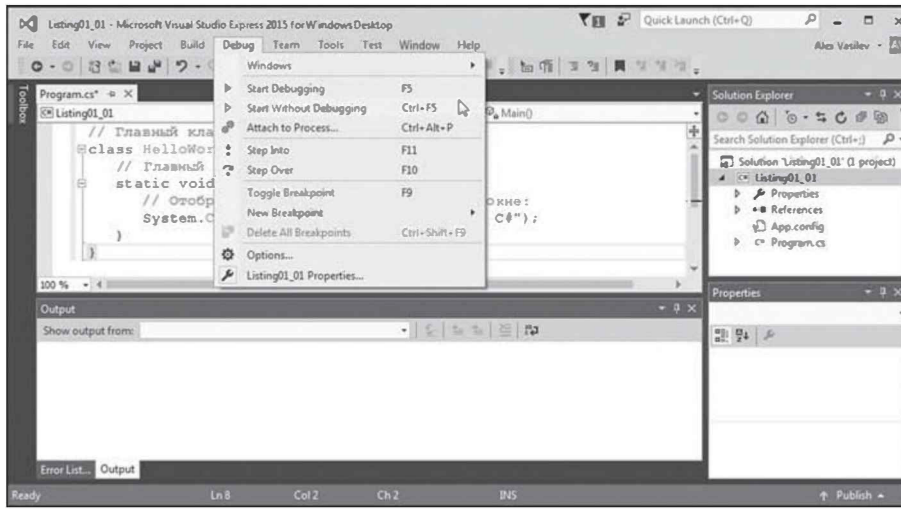


Рис. 1.5. Запуск программы на выполнение

Если компиляция прошла успешно, то в консольном окне появляется результат выполнения программы (в данном случае — сообщение), как показано на рис. 1.6.

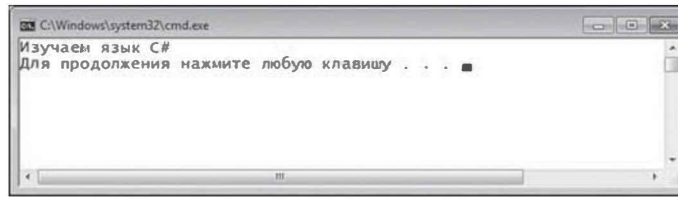


Рис. 1.6. Результат выполнения программы отображается в консольном окне

НА ЗАМЕТКУ

Кроме сообщения, которое выводит в консольное окно программа, там также появляется сообщение **Для продолжения нажмите любую клавишу**. Это сообщение добавляется автоматически операционной системой и к программе не имеет никакого отношения.

ПОДРОБНОСТИ

Если для компилирования и запуска воспользоваться командой **Start Debugging** из меню **Debug** или пиктограммой на панели инструментов, то программа будет запущена, но консольное окно закроется сразу после завершения выполнения программы. Все происходит настолько быстро, что пользователь, как правило, не успевает прочитать содержимое консольного окна. Если читатель столкнется с подобной проблемой, то последней командой в программу можно добавить инструкцию `System.Console.ReadLine()` (это должна быть последняя команда в методе `Main()`).

Не исключено, что для вывода кириллического текста в консольное окно придется выполнить некоторые настройки и для консольного окна. Для этого в левом верхнем углу консольного окна следует щелкнуть правой кнопкой мыши и в раскрывшемся списке выбрать команду **Свойства** (для выполнения настроек открытого консольного окна) или **Умолчания** (для выполнения настроек, используемых по умолчанию). В открывшемся окне **Свойства консольного окна** следует задать шрифт, который поддерживает кириллические символы, размер шрифта, фон для окна консоли и выполнить некоторые другие настройки.

Пространство имен

> Не надо громких слов, они потрясают воздух, но не собеседника.
>
> *из к/ф «Формула любви»*

Мы можем немного модифицировать код нашей первой программы. Дело в том, что команда вида `System.Console.WriteLine()`, в которой явно указано пространство имен `System`, не очень удобна — она слишком длинная. Вообще, длинные команды в C# — самое обычное дело. Но в данном случае мы имеем дело с длинной командой, которую можно немножко сократить. Для этого в программу в самом начале (до описания класса) следует добавить инструкцию `using System` (инструкция в программе заканчивается точкой с запятой). Инструкция означает, что в программе используется пространство имен `System`. Как следствие, мы можем обращаться к классам из этого пространства имен без явного указания названия пространства `System`. В итоге инструкция `System.Console.WriteLine()`

трансформируется в инструкцию `Console.WriteLine()` (пространство имен `System` явно не указывается). Новая версия нашей первой программы представлена в листинге 1.2.

Листинг 1.2. Использование пространства имен

```
// Использование пространства имен System:
using System;
// Главный класс программы:
class HelloWorld{
   // Главный метод программы:
   static void Main(){
      // Отображение сообщения в консольном окне:
      Console.WriteLine("Изучаем язык C#");
   }
}
```

Результат выполнения этой программы точно такой же, как и в предыдущем случае.

НА ЗАМЕТКУ

Если при запуске программы консольное окно закрывается слишком быстро, то программный код метода `Main()` можно завершить командой `Console.ReadLine()`. В таком случае консольное окно не будет закрыто, пока пользователь не нажмет клавишу <Enter>. Если вместо команды `Console.ReadLine()` использовать команду `Console.ReadKey()`, то для закрытия консольного окна достаточно будет нажать любую клавишу.

Вообще же статический метод `ReadLine()` из класса `Console` предназначен для считывания текстовой строки, которую пользователь вводит с помощью клавиатуры. Статический метод `ReadKey()` из класса `Console` используется для считывания символа, введенного пользователем с помощью клавиатуры.

В дальнейшем мы будем использовать `using`-инструкции для подключения пространств имен. Сразу отметим, что таких инструкций в программе может быть несколько (то есть в программе одновременно разрешается использовать несколько пространств имен).

Программа с диалоговым окном

> Ну, барин, ты задачи ставишь! За десять ден одному не справиться. Тут помощник нужен — хомо сапиенс.
>
> *из к/ф «Формула любви»*

Следующая программа, которую мы рассмотрим, выводит сообщение в диалоговом окне: при запуске программы на выполнение появляется диалоговое окно, содержащее текст (сообщение). В окне будет кнопка **ОК**, и щелчок на ней приводит к закрытию окна и завершению выполнения программы. Для отображения окна мы воспользуемся статическим методом `Show()` из класса `MessageBox`. Класс описан в пространстве имен `Forms`, которое входит в пространство имен `Windows`, которое, в свою очередь, входит в пространство имен `System`. Эта иерархия отображается инструкцией `System.Windows.Forms`. Соответственно, программный код начинается инструкцией `using System.Windows.Forms`, подключающей пространство имен `Forms`. Это дает возможность использовать класс `MessageBox` и его метод `Show()`.

ПОДРОБНОСТИ

Мало добавить в программный код инструкцию `using System.Windows.Forms`. Необходимо добавить ссылку на соответствующее пространство имен в проект. Для этого в окне **Solution Explorer** (по умолчанию отображается в правом верхнем углу окна среды разработки) необходимо щелкнуть правой кнопкой мыши на узле проекта и в раскрывшемся контекстном меню выбрать подменю **Add**, а в нем команду **Reference**. В результате откроется окно, в котором следует выбрать ссылку для включения в проект. Вся эта процедура описывается и иллюстрируется далее.

Если при вызове метода `Show()` ему передать аргументом текстовое значение, то появится диалоговое окно, содержащее данный текст. Как выглядит код программы, в которой отображается диалоговое окно, показано в листинге 1.3.

Листинг 1.3. Отображение диалогового окна

```
// Использование пространства имен:
using System.Windows.Forms;
```

```
// Класс с главным методом программы:
class DialogDemo{
   // Главный метод программы:
   static void Main(){
      // Отображение диалогового окна:
      MessageBox.Show("Продолжаем изучать C#");
   }
}
```

Кроме инструкции подключения пространства имен в начале программы, в главном методе программы имеется команда `MessageBox.Show("Продолжаем изучать C#")`. Как отмечалось ранее, в результате выполнения этой команды появится диалоговое окно. Но способ создания проекта в данном случае немного отличается от того, что рассматривался выше. Далее описываются основные этапы реализации программы.

> **НА ЗАМЕТКУ**
>
> Проект можно создавать и выполнять его настройки по-разному. Далее мы иллюстрируем один из возможных способов.

Итак, на этот раз будем создавать пустой проект — для этого в окне создания проекта **New Project** в качестве типа проекта выбираем **Empty Project**, как показано на рис. 1.7.

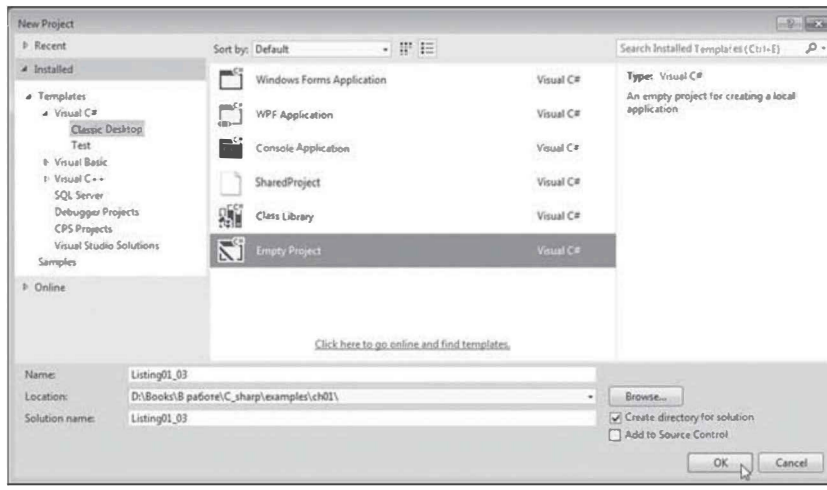


Рис. 1.7. Создание пустого проекта

ПОДРОБНОСТИ

Для создания проекта в меню **File** выбираем команду **New Project**. В окне **New Project** в левом списке в узле **Visual C#** выбираем пункт **Classic Desktop**.

После того как объект создан, в него нужно добавить файл, в который мы введем программный код. Для этого во внутреннем окне **Solution Explorer** в правом верхнем углу окна среды разработки выбираем узел проекта и щелкаем правой кнопкой мыши. Откроется контекстное меню для проекта, в котором в подменю **Add** выбираем команду **New Item**, как показано на рис. 1.8.

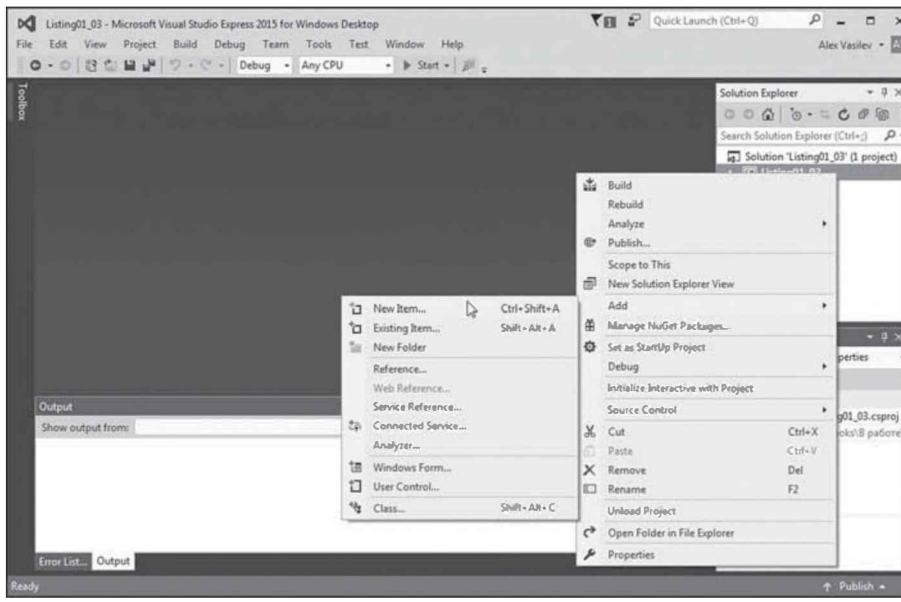


Рис. 1.8. Добавление элемента в созданный пустой проект

НА ЗАМЕТКУ

Вместо контекстного меню можем воспользоваться командой **Add New Item** в меню **Project**.

В результате откроется окно **Add New Item** для добавления элемента в проект. Окно показано на рис. 1.9.

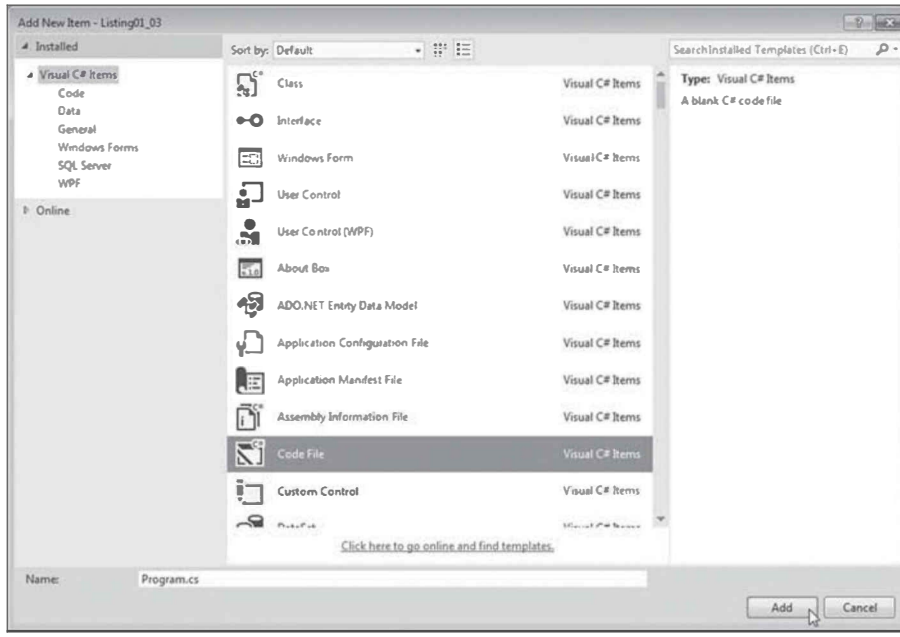


Рис. 1.9. Добавление в проект файла с программным кодом

В этом окне для типа добавляемого элемента выбираем **Code File**, а в поле **Name** указываем название для файла (в данном примере мы назовем файл `Program.cs`). После щелчка на кнопке **Add** файл добавляется в проект. После этого в проекте есть файл, в который мы собираемся ввести программный код. Как выглядит проект с добавленным в него файлом (еще пустым), показано на рис. 1.10.

ПОДРОБНОСТИ

В окне **Solution Explorer** отображается «начинка» проекта. В частности, после добавления файла в проект в узле проекта отображается пункт с названием добавленного файла. Чтобы содержимое файла открылось в окне редактора кодов, достаточно в окне **Solution Explorer** выделить мышью пункт с названием файла.

Если окно **Solution Explorer** по каким-то причинам не отображается, нужно в меню **View** выбрать команду **Solution Explorer**.

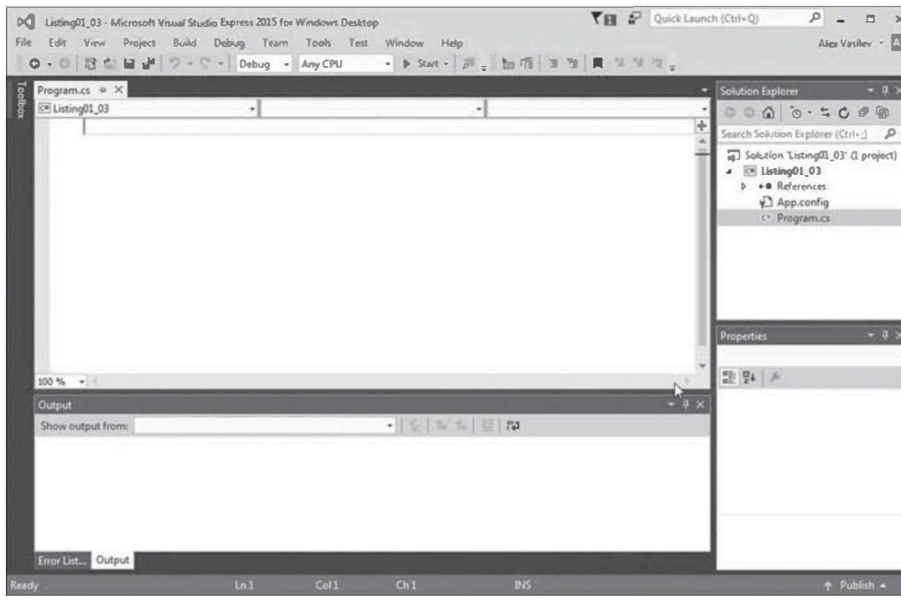


Рис. 1.10. Проект с файлом для ввода кода программы

После ввода программного кода окно среды разработки с открытым в ней проектом будет выглядеть так, как показано на рис. 1.11.

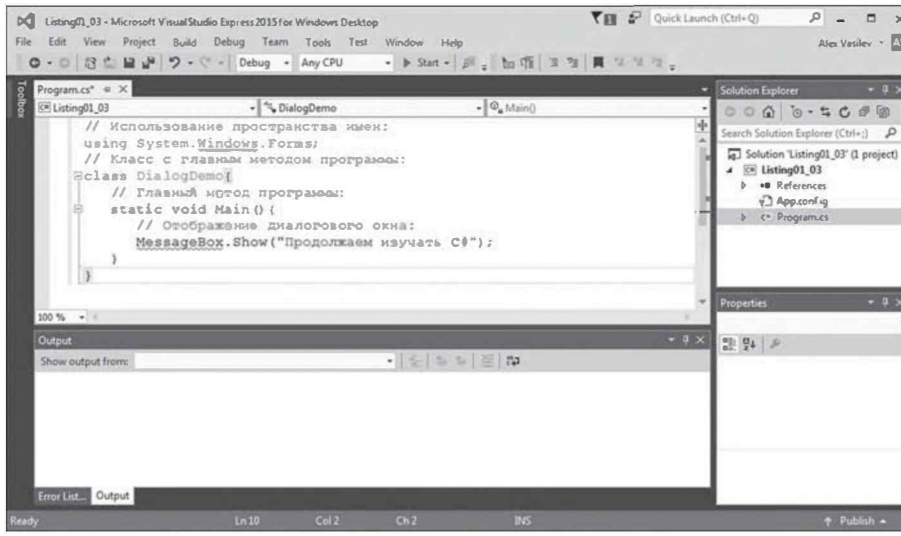


Рис. 1.11. Программный код введен в окно редактора кодов

Хотя мы уже ввели программный код, проект к работе еще не готов. В него необходимо добавить ссылку на пространство имен. Для

этого в контекстном меню проекта в подменю **Add** выбираем команду **Reference**. Ситуация проиллюстрирована на рис. 1.12.

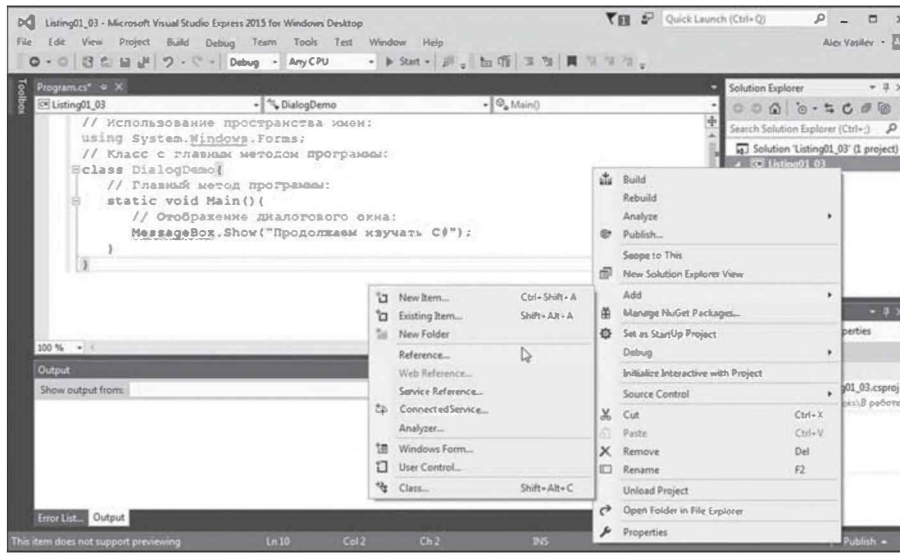


Рис. 1.12. Добавление ссылок в проект

Откроется окно **Reference Manager**, в котором мы находим пункт **System.Windows.Forms** и устанавливаем возле него флажок (рис. 1.13).

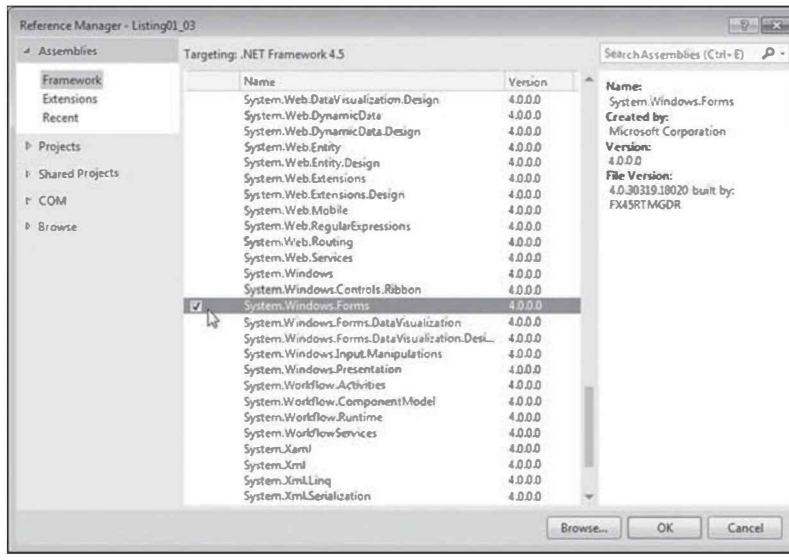


Рис. 1.13. Выбор ссылки для добавления в проект

Для подтверждения щелкаем кнопку **OK**.

ПОДРОБНОСТИ

Пока ссылка в проект не добавлена, команды, в которых есть обращение к неподключенному пространству имен и классам из этого пространства имен, подчеркиваются красной волнистой чертой, что означает наличие ошибок в программном коде. Причина в том, что, пока соответствующая ссылка не добавлена в проект, использованные идентификаторы компилятору «не известны». После подключения ссылки красное подчеркивание должно исчезнуть.

Если мы используем пространство имен `System`, ссылка в проекте на это пространство тоже должна быть. Но при создании консольного приложения ссылка на пространство имен `System` (и некоторые другие) в проект добавляется автоматически. Поэтому раньше у нас не было необходимости выполнять описываемые настройки.

Также стоит заметить, что добавить ссылку в проект можно с помощью команды **Add Reference** из меню **Project**.

Технически проект готов к использованию. Осталось один маленький «штрих» добавить. А именно, мы установим тип приложения. Для этого в контекстном меню проекта выбираем команду **Properties** (рис. 1.14) или выбираем одноименную команду в меню **Project**.

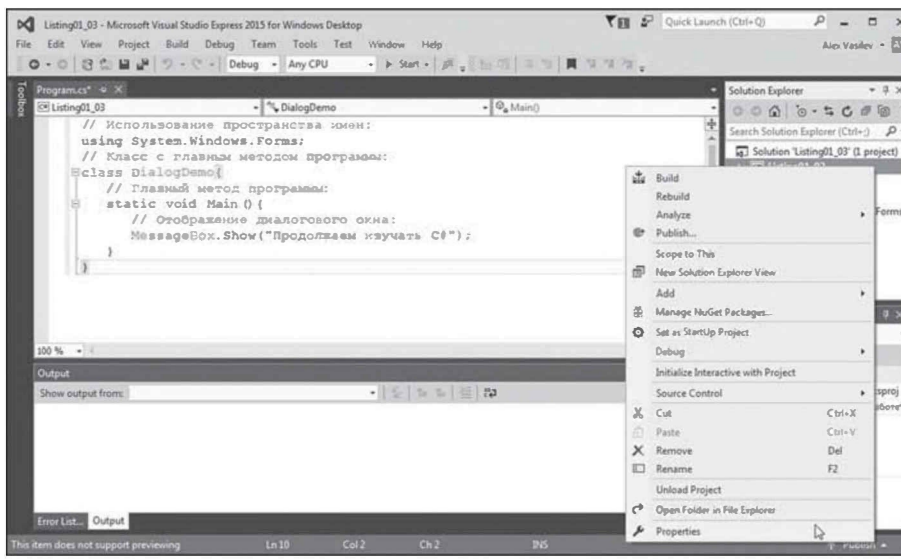


Рис. 1.14. Выбор команды ***Properties*** в контекстном меню проекта для перехода к вкладке свойств проекта

Во внутреннем окне редактора кодов откроется вкладка свойств проекта. Она показана на рис. 1.15.

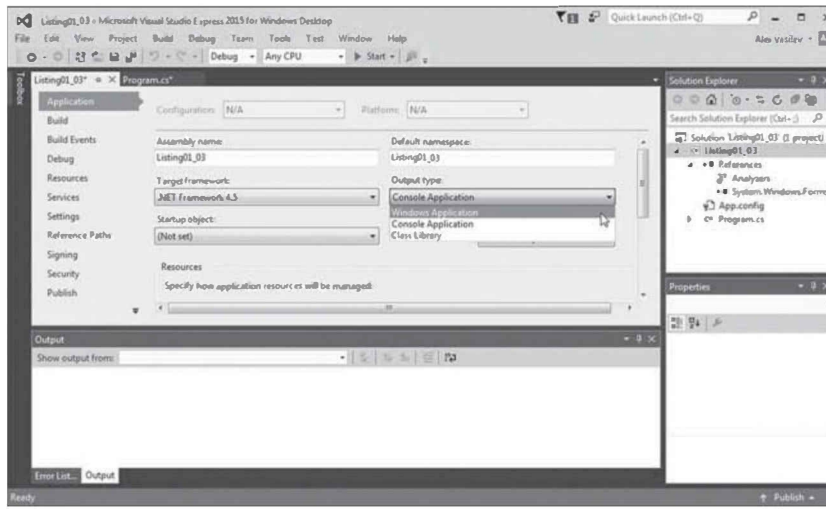


Рис. 1.15. Выбор типа приложения на вкладке свойств проекта

На вкладке свойств проекта следует выбрать раздел **Application**, а в раскрывающемся списке **Output type** устанавливаем значение **Windows Application** (вместо используемого по умолчанию значения **Console Application**). Чтобы закрыть вкладку свойств проекта, щелкаем системную пиктограмму (рис. 1.16).

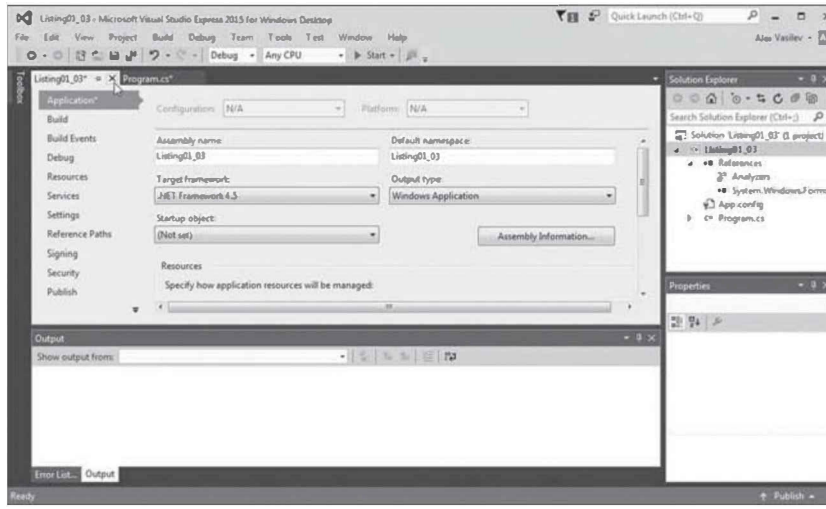


Рис. 1.16. Закрытие вкладки свойств проекта

Для компилирования и запуска программы на выполнение выбираем, например, команду **Start Debugging** в меню **Debag** (рис. 1.17).

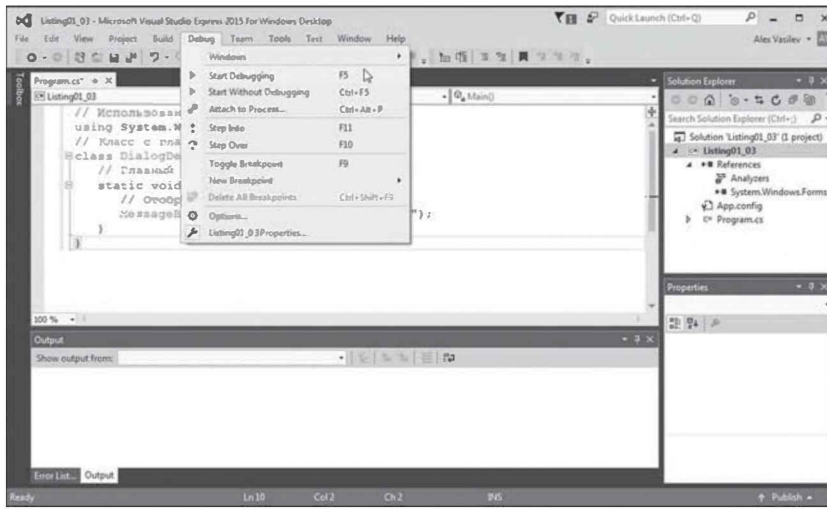


Рис. 1.17. Запуск приложения на выполнение с помощью команды *Start Debugging* в меню *Debug*

Если программа компилируется без ошибок, появляется диалоговое окно, представленное на рис. 1.18.

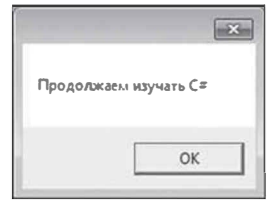


Рис. 1.18. При выполнении программы отображается окно с сообщением

Щелчок на кнопке **OK** в этом окне (или на системной пиктограмме в строке названия окна) приводит к закрытию окна и завершению выполнения программы.

ПОДРОБНОСТИ

Если не изменить тип приложения на **Windows Application**, то при запуске программы кроме диалогового окна откроется еще и консольное окно. Это не страшно, но не очень эстетично.

НА ЗАМЕТКУ

Вместо добавления файла с кодом в пустой проект можно было добавить класс. В этом случае в контекстном меню проекта в подменю **Add** вместо команды **New Item** выбирают команду **Class** (см. рис. 1.8). Но, строго говоря, можно было создавать не пустой проект, а консольное приложение. Тогда добавлять в проект файл для ввода программного кода не придется — такой файл в проект добавляется автоматически. Все прочие настройки выполняются так, как описано выше.

Настройка вида диалогового окна

> Да, это от души. Замечательно. Достойно восхищения.
>
> *из к/ф «Формула любви»*

Окно, которое отображалось в предыдущем примере, было очень простым (см. рис. 1.18). Оно было настолько простым, что не содержало даже названия. Эту досадную оплошность легко исправить. Настройка диалогового окна выполняется передачей методу `Show()` из класса `MessageBox` дополнительных аргументов. Ниже описывается назначение аргументов метода `Show()`.

- Первый аргумент, как мы уже знаем, определяет сообщение, отображаемое в окне.
- Если передать методу второй текстовый аргумент, то такой аргумент будет определять заголовок диалогового окна (отображается в строке названия).
- Третий аргумент определяет тип пиктограммы, отображаемой в диалоговом окне (по умолчанию пиктограмма в окне не отображается).
- Четвертый аргумент определяет кнопки, которые отображаются в диалоговом окне (по умолчанию такая кнопка одна, и это кнопка **ОК**).

Таким образом, методу `Show()` можно передавать от одного до четырех аргументов. С первыми двумя аргументами (сообщение и заголовок окна) все просто — это текстовые значения. Третий и четвертый аргументы не так просты, а именно: третий аргумент должен быть константой из перечисления `MessageBoxButtons`.

ПОДРОБНОСТИ

Перечисления рассматриваются немного позже. Пока же просто отметим, что перечисление представляет собой набор констант (фиксированных, неизменных значений). Каждая константа имеет название. Название константы указывается вместе с названием перечисления: после имени перечисления ставится точка, и затем указывается название константы. Например, константа `OK` из перечисления `MessageBoxButtons`, переданная третьим аргументом методу `Show()`, означает, что в диалоговом окне должна отображаться кнопка с названием **OK**. Константа `OK` передается аргументом в виде инструкции `MessageBoxButtons.OK`.

Константы из перечисления `MessageBoxButtons`, определяющие количество и названия отображаемых в диалоговом окне кнопок, перечислены в табл. 1.1.

Табл. 1.1. Константы для определения кнопок в диалоговом окне

Константа	Онисание
`OK`	Отображение кнопки **OK**
`OKCancel`	Отображение кнопок **OK** и **Cancel (Отмена)**
`YesNo`	Отображение кнопок **Yes (Да)** и **No (Нет)**
`YesNoCancel`	Отображение кнопок **Yes (Да)**, **No (Нет)** и **Cancel (Отмена)**
`RetryCancel`	Отображение кнопок **Retry (Повтор)** и **Cancel (Отмена)**
`AbortRetryIgnore`	Отображение кнопок **Abort (Прервать)**, **Retry (Повтор)** и **Ignore (Пропустить)**

Еще одна константа, на этот раз из перечисления `MessageBoxIcon`, определяет тип пиктограммы, отображаемой в диалоговом окне. Константы, используемые в качестве четвертого аргумента метода `Show()`, представлены в табл. 1.2.

Табл. 1.2. Константы для определения пиктограммы в диалоговом окне

Константа	Пиктограмма
`Asterisk`	Отображается пиктограмма с изображением прописной буквы **i** внутри синего круга
`Error`	Буква **X** белого цвета внутри красного круга
`Exclamation`	Восклицательный знак **!** внутри желтого треугольника
`Hand`	Буква **X** белого цвета внутри красного круга

`Information`	Отображается пиктограмма с изображением прописной буквы **i** внутри синего круга
`None`	Пиктограмма отсутствует
`Question`	Знак вопроса **?** внутри синего круга
`Stop`	Буква **X** белого цвета внутри красного круга
`Warning`	Восклицательный знак **!** внутри желтого треугольника

Как и в случае констант из перечисления `MessageBoxButtons` (третий аргумент), константа из перечисления `MessageBoxIcon` указывается вместе с названием перечисления. Например, константа `Information` передается аргументом методу `Show()` в виде инструкции `MessageBoxIcon.Information`. Небольшой пример использования констант из перечислений `MessageBoxButtons` и `MessageBoxIcon` представлен в листинге 1.4.

Листинг 1.4. Диалоговое окно с названием и пиктограммой

```
using System.Windows.Forms;
class AnotherDialogDemo{
   static void Main(){
      // Отображение диалогового окна:
      MessageBox.Show("Всем привет!", // Сообщение
         "Окно с названием",          // Название окна
         MessageBoxButtons.OK,        // Кнопки (одна OK)
         MessageBoxIcon.Information   // Пиктограмма
      );
   }
}
```

При выполнении программы отображается диалоговое окно с сообщением. Как выглядит окно, показано на рис. 1.19.

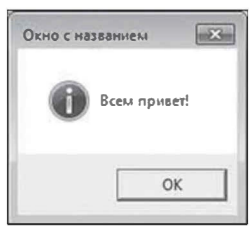


Рис. 1.19. Диалоговое окно с пиктограммой и названием

Особенность этого окна в том, что, помимо сообщения и кнопки **OK**, у окна есть название **Окно с названием** и информационная пиктограмма (буква **i** внутри синего круга). Окно, которое отображалось в предыдущем примере (см. рис. 1.18), не имело ни пиктограммы, ни названия. Для определения заголовка диалогового окна мы передаем вторым аргументом методу `Show()` текст `"Окно с названием"` (см. листинг 1.4). Третий аргумент `MessageBoxButtons.OK` означает, что в диалоговом окне отображается всего одна кнопка **OK**. Четвертый аргумент `MessageBoxIcon.Information` определяет пиктограмму, отображаемую в диалоговом окне.

ПОДРОБНОСТИ

В нашем случае диалоговое окно содержит всего одну кнопку. В принципе, можно сделать так, чтобы кнопок было несколько (две или три). В таком случае пользователь может нажать одну из кнопок в диалоговом окне, и нам важно уметь определять, какая именно кнопка была нажата. Такая возможность есть.

Метод `Show()` возвращает результат: инструкция с вызовом метода имеет значение. Это значение — одна из констант перечисления `DialogResult`. Константы определяют кнопку, нажатую пользователем: `Yes` (нажата кнопка **Yes**, или **Да**), `No` (нажата кнопка **No**, или **Нет**), `OK` (нажата кнопка **OK**), `Cancel` (нажата кнопка **Cancel**, или **Отмена**), `Retry` (нажата кнопка **Retry**, или **Повтор**), `Abort` (нажата кнопка **Abort**, или **Прервать**), `Ignore` (нажата кнопка **Ignore**, или **Пропустить**). Как использовать эти значения для определения нажатой в окне кнопки, мы рассмотрим немного позже.

Окно с полем ввода

> Ну, ежели, конечно, кроме железных предметов, еще и фарфор можете унотребить — тогда... нросто слов нет.
>
> *из к/ф «Формула любви»*

В следующем примере мы научимся отображать диалоговое окно с полем ввода. С помощью такого окна в программу можно вводить данные, которые программа сможет использовать.

Диалоговое окно с полем ввода отображается так же просто, как и диалоговое окно с сообщением. Правда, для отображения окна с полем ввода нам придется прибегнуть к средствам языка Visual Basic. Дело в том, что в языке C# нет встроенных средств для отображения окна с полем ввода. Зато такое окно можно отобразить с привлечением средств языка Visual Basic. Технология .Net Framework позволяет нам прибегнуть к помощи библиотеки Visual Basic при написании программного кода на языке C#. Чтобы воспользоваться этой возможностью, мы должны подключить пространство имен `Microsoft.VisualBasic`. Если пространство подключено, то в программе будет доступен класс `Interaction`, в котором есть статический метод `InputBox()`. Этот метод отображает диалоговое окно с полем ввода, а результатом возвращает текст, введенный пользователем. Небольшой пример программы, в которой использованы описанные выше утилиты для отображения диалогового окна с полем ввода, представлен в листинге. 1.5.

Листинг 1.5. Отображение окна с полем ввода

```
// Используем ресурсы Visual Basic:
using Microsoft.VisualBasic;
using System.Windows.Forms;
class InputDialogDemo{
   static void Main(){
      // Текстовая переменная:
      string name;
      // Отображение окна с полем ввода:
      name=Interaction.InputBox(
         "Как Вас зовут?",       // Текст над полем ввода
         "Давайте познакомимся…" // Название окна
      );
      // Еще одна текстовая переменная:
      string txt="Очень приятно, "+name+"!";
      // Окно с сообщением:
      MessageBox.Show(txt,"Знакомство состоялось");
   }
}
```

При запуске программы сначала появляется окно, представленное на рис. 1.20.

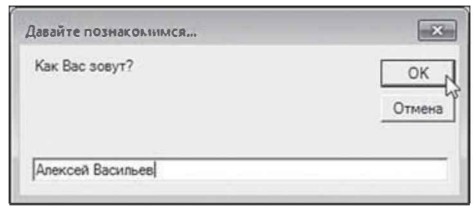


Рис. 1.20. Диалоговое окно с полем ввода

Окно имеет название **Давайте познакомимся...**, определяемое в программе, и текст **Как Вас зовут?**, также определяемый в программе. В поле диалогового окна следует ввести текст и щелкнуть кнопку **OK** (см. рис. 1.20). После этого появляется диалоговое окно с сообщением, причем сообщение содержит текст, введенный ранее пользователем в поле предыдущего диалогового окна. Как может выглядеть окно с сообщением, показано на рис. 1.21.

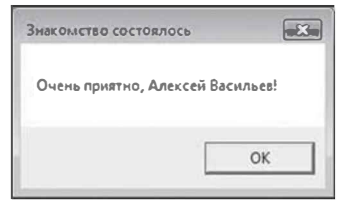


Рис. 1.21. Диалоговое окно с сообщением, содержащим введенный пользователем текст

Теперь проанализируем программный код, который позволяет получить такие результаты. В начале программы подключаются пространства имен `Microsoft.VisualBasic` и `System.Windows.Forms`. Первое из них необходимо для использования статического метода `InputBox()` из класса `Interaction`, отображающего окно с полем ввода, а второе пространство имен необходимо для использования статического метода `Show()` из класса `MessageBox`.

НА ЗАМЕТКУ

Напомним, что мало добавить в программный код инструкции подключения пространств имен. Необходимо также добавить

соответствующие ссылки в проект. Для этого следует в меню **Project** выбрать команду **Add Reference** и в открывшемся окне установить флажки у пунктов с названиями подключаемых пространств имен.

В главном методе программы есть команда `string name`. Этой командой объявляется переменная `name`, относящаяся к текстовому типу `string`. Проще говоря, значением переменной `name` может быть текст.

Переменная — это блок памяти, к которому обращаются по имени (название переменной). Переменную удобно представлять в виде ячейки с надписью на дверце. В эту ячейку можно что-то положить (записать значение) или посмотреть, что там находится (прочитать значение). У каждой переменной есть тип. Тип переменной влияет на объем памяти, выделяемой под переменную, а также на то, какие значения можно записывать в переменную и какие операции с этой переменной можно выполнять.

Каждая переменная в языке C# должна быть объявлена. Для этого указывается тип переменной и ее название. В данном случае в качестве идентификатора типа указано ключевое слово `string`, означающее, что переменная является текстовой: значением такой переменной разрешается присвоить текст. Объявлять переменные можно фактически в любом месте программы (главного метода), но обязательно до того, как переменная первый раз используется.

ПОДРОБНОСТИ

Идентификатор `string` является псевдонимом (или синонимом) для инструкции `System.String`. Это обращение к классу `String` из пространства имен `System`. Если пространство имен `System` подключено, то вместо идентификатора `string` можно использовать название класса `String`.

Класс `String` предназначен для реализации текста. Текст реализуется в виде объекта класса `String`. Но то, как выше описывался способ реализации переменных, относится к базовым типам, а не к объектам. Другими словами, способ реализации объектов сложнее, чем было описано выше. Тем не менее пока что это не столь важно, и мы можем думать о текстовой переменной как об обычной переменной базового типа. Базовые типы данных, классы и объекты обсуждаются позже.

Для отображения диалогового окна с полем ввода из класса `Interaction` вызывается статический метод `InputBox()`. Аргументами методу передается текст `"Как Вас зовут?"` (первый аргумент) и `"Давайте познакомимся..."` (второй аргумент). Первый аргумент метода определяет текст, отображаемый в диалоговом окне сверху над полем ввода, в то время как второй аргумент задает название окна.

Метод `InputBox()` возвращает результат. Это позволяет отождествлять инструкцию вызова метода с некоторым значением. Для метода `InputBox()` возвращаемое значение — это введенный пользователем в поле диалогового окна текст. Данное значение присваивается переменной `name`. Поэтому в результате выполнения команды `name=Interaction.InputBox("Как Вас зовут?","Давайте познакомимся...")` открывается диалоговое окно с полем ввода, а переменной `name` значением присваивается (после того как пользователь нажмет кнопку **OK**) текст из поля ввода.

ПОДРОБНОСТИ

В C# оператором присваивания является символ равенства `=`. Выражение вида `переменная=значение` обрабатывается так: в `переменную`, указанную слева от оператора присваивания, записывается `значение`, указанное справа от оператора присваивания.

Командой `string txt="Очень приятно, "+name+"!"` объявляется еще одна текстовая переменная, которая называется `txt`. При объявлении переменной ей сразу присваивается значение. Так можно делать. Значением, присваиваемым переменной `txt`, указано выражение `"Очень приятно, "+name+"!"`. Это сумма (использован оператор сложения `+`) трех текстовых значений: к текстовому литералу `"Очень приятно, "` прибавляется значение переменной `name`, и к тому, что получилось, прибавляется текстовый литерал `"!"` (состоящий всего из одного символа, но все равно это текст). Если операция сложения применяется к текстовым значениям, то происходит формирование новой текстовой строки путем объединения (конкатенации) складываемых фрагментов. В данном случае переменной `txt` присваивается текст, получающийся объединением текста `"Очень приятно, "`, значения текстовой переменной `name` (то, что ввел пользователь) и текста `"!"`. После того как значение переменной `txt` определено, переменная передается первым аргументом методу `Show()` из класса `MessageBox`. Вторым аргументом передается текст `"Знакомство состоялось"`,

определяющий заголовок диалогового окна. В результате выполнения команды `MessageBox.Show(txt,"Знакомство состоялось")` появляется диалоговое окно с сообщением, содержащим введенный пользователем текст.

Консольный ввод

— Вы получите то, что желали, согласно намеченным контурам.
— К черту контуры! Я их уже ненавижу.

из к/ф «Формула любви»

Информацию можно вводить не только через окно с полем ввода, но и через консольные устройства: набирая значение с помощью клавиатуры (с отображением в консольном окне). Далее мы рассмотрим пример, в котором показано, как в программе может быть реализован консольный ввод. Программа похожа на предыдущую (см. листинг 1.5): пользователь вводит текст, и затем появляется сообщение, содержащее этот текст. Но в предыдущем примере текст вводился в поле диалогового окна, и сообщение появлялось в диалоговом окне. Теперь для ввода и вывода информации мы используем консольное окно. Нам понадобится статический метод `ReadLine()` из класса `Console` для считывания введенного пользователем текста. Для вывода сообщений в консольное окно мы используем статические методы `Write()` и `WriteLine()` из класса `Console`. Различие между методами `Write()` и `WriteLine()` в том, что методом `Write()` в консольном окне отображается значение аргумента (например, текст) и курсор остается в той же строке. То есть следующее сообщение появится в той же строке, в которую выводилось предыдущее сообщение. Методом `WriteLine()` отображается значение аргумента в консольном окне, и по завершении отображения выполняется переход к новой строке: следующее сообщение появится в новой строке. Далее рассмотрим программный код в листинге 1.6.

НА ЗАМЕТКУ

В данном случае мы создаем консольный проект. Для компилирования и запуска программы на выполнение используем команду **Start Without Debugging** из меню **Debug** или нажимаем комбинацию клавиш <Ctrl>+<F5>.

Листинг 1.6. Консольный ввод

```
using System;
class InputConsoleDemo{
   static void Main(){
      // Текстовая переменная:
      string name;
      // Заголовок консольного окна:
      Console.Title="Давайте познакомимся…";
      // Сообщение в консоли:
      Console.Write("Как Вас зовут?");
      // Считывание текста:
      name=Console.ReadLine();
      // Еще одна текстовая переменная:
      string txt="Очень приятно, "+name+"!";
      // Заголовок консольного окна:
      Console.Title="Знакомство состоялось";
      // Сообщение в консоли:
      Console.WriteLine(txt);
   }
}
```

В программе используется класс `Console`, поэтому мы подключаем пространство имен `System`, к которому относится этот класс.

ПОДРОБНОСТИ

Также мы используем класс `String` из пространства имен `System` для реализации текстовых переменных. Но вместо названия класса `String` при объявлении текстовых переменных задействован идентификатор `string` как псевдоним для инструкции `System.String`. В последующих примерах мы также в основном будем использовать идентификатор `string`.

В главном методе программы командой `string name` объявляется текстовая переменная `name`. Эта переменная используется для запоминания текста, введенного пользователем. Командой `Console.`

`Title="Давайте познакомимся..."` задается заголовок для консольного окна.

ПОДРОБНОСТИ

У класса `Console` есть свойство `Title`, определяющее название, отображаемое в консольном окне. По умолчанию при запуске программы на выполнение в названии консольного окна отображается путь к файлу программы. Присвоив значение свойству, мы определяем название, которое отображается в строке названия консольного окна. Более детально свойства обсуждаются в контексте изучения классов и объектов.

Командой `Console.Write("Как Вас зовут?")` в консоли печатается сообщение. Курсор остается в той же строке. Ситуация проиллюстрирована на рис. 1.22.

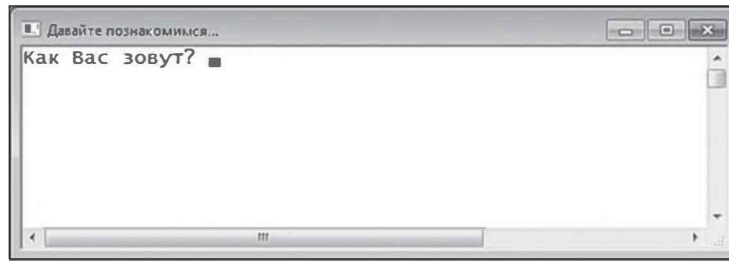


Рис. 1.22. Сообщение в консольном окне и ожидание ввода пользователя

Программа ожидает, пока пользователь введет значение (текст). Следует ввести текст (имя пользователя) и нажать клавишу <Enter>. На рис. 1.23 показана ситуация, когда пользователь ввел текст, но еще не нажал клавишу <Enter>.

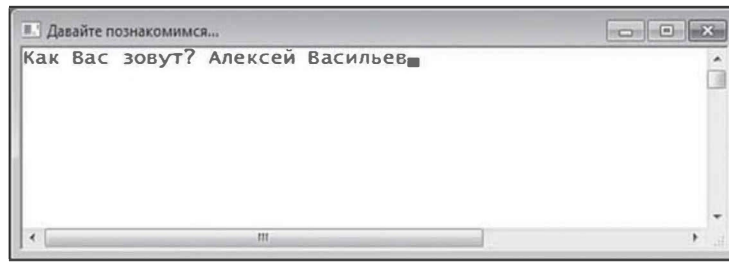


Рис. 1.23. Консольное окно с введенным текстом перед нажатием клавиши <Enter>

Считывание введенного пользователем текста выполняется командой `name=Console.ReadLine()`. Здесь из класса `Console` вызывается статический метод `ReadLine()`. Метод считывает введенное пользователем текстовое значение и возвращает это значение результатом. Текст запоминается с помощью текстовой переменной `name`. Еще одна текстовая переменная объявляется и инициализируется командой `string txt="Очень приятно, "+name+"!"`. Значение переменной `txt` формируется объединением текста `"Очень приятно, "`, значения текстовой переменной `name` и текста `"!"`.

НА ЗАМЕТКУ

Инициализация переменной означает первое (после объявления или во время объявления) присваивание значения этой переменной.

После этого командой `Console.Title="Знакомство состоялось"` консольному окну присваивается новое название, а затем с помощью команды `Console.WriteLine(txt)` в консольном окне отображается значение переменной `txt`. Как может выглядеть в этом случае консольное окно, показано на рис. 1.24.

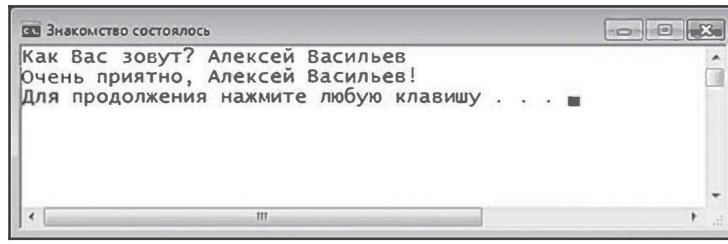


Рис. 1.24. Консольное окно после отображения сообщения с текстом, введенным пользователем

НА ЗАМЕТКУ

Сообщение **Для продолжения нажмите любую клавишу...**, которое появляется в консольном окне в последней строчке, не имеет отношения к программе и добавляется автоматически.

Следовательно результат выполнения программы может быть таким, как показано ниже (жирным шрифтом выделено значение, введенное пользователем).

Результат выполнения программы (из листинга 1.6)

```
Как Вас зовут? Алексей Васильев
Очень приятно, Алексей Васильев!
```

В этом и рассмотренном ранее примере с вводом значения через окно с полем ввода мы считывали текст. Но часто возникает необходимость считывать значения и других типов.

Считывание чисел

Ни одной буквочки! Ни одной «мэйд ин...».

из к/ф «Кин-дза-дза»

При считывании введенного пользователем значения с помощью метода `ReadLine()` (при консольном вводе) или метода `InputBox()` (при вводе через поле в диалоговом окне) значение, введенное пользователем, считывается как текст — даже если это число. Проще говоря, если пользователь введет, например, целочисленное значение, то считано оно будет как текст, состоящий из цифр (это называется *текстовое представление числа*). Чтобы «извлечь» из такого текста число, необходимо использовать специальные методы. Для получения целочисленного значения на основе его текстового представления используют статический метод `Parse()` из структуры `Int32` (структура относится к пространству имен `System`).

НА ЗАМЕТКУ

В языке C# структура является аналогом класса. Во всяком случае, различия между ними в основном «технические». Мы будем детально обсуждать и классы, и структуры.

Аргументом методу передается текстовое представление числа, а результатом метод возвращает собственно число, «спрятанное» в тексте. Небольшой пример считывания целочисленного значения с помощью диалогового окна с полем ввода представлен в листинге 1.7. Программа, которая представлена там, при выполнении отображает диалоговое окно с полем ввода, в которое пользователю предлагается ввести год своего рождения. Год рождения считывается, и на основе этой информации вычисляется возраст пользователя. А теперь рассмотрим код программы.

Листинг 1.7. Считывание числа

```
using System;
using Microsoft.VisualBasic;
using System.Windows.Forms;
class EnteringInteger{
   static void Main(){
      // Текстовые переменные:
      string res, txt;
      // Целочисленные переменные:
      int year=2017, age, born;
      // Отображение окна с полем ввода:
      res=Interaction.InputBox("В каком году Вы родились?","Год рождения");
      // Преобразование текста в число:
      born=Int32.Parse(res);
      // Вычисление возраста:
      age=year-born;
      txt="Тогда Вам "+age+" лет";
      // Окно с сообщением:
      MessageBox.Show(txt,"Возраст");
   }
}
```

В программе подключаются пространства имен `System`, `Microsoft.VisualBasic` и `System.Windows.Forms`. В главном методе объявляются текстовые переменные `res` и `txt` (команда `string res, txt`). Также объявляются три целочисленные переменные `year`, `age` и `born` (команда `int year=2017, age, born`), причем переменной `year` сразу присваивается значение `2017`. Для определения типа мы использовали идентификатор `int`, обозначающий целочисленный тип: переменные такого типа значением могут принимать целые числа.

НА ЗАМЕТКУ

Можно объявлять сразу несколько переменных одного типа. В таком случае указывается идентификатор типа переменных, и затем через

запятую перечисляются эти переменные. В случае необходимости части из них (или всем переменным) могут присваиваться значения.

Командой `res=Interaction.InputBox("В каком году Вы родились?","Год рождения")` отображается окно с полем ввода. Первый аргумент метода `InputBox()` определяет текст, отображаемый в диалоговом окне сверху над полем ввода, а второй аргумент метода задает название диалогового окна. Как выглядит окно, показано на рис. 1.25.

Рис. 1.25. Диалоговое окно с полем для ввода целого числа

Введенное пользователем число в виде текстового значения запоминается с помощью переменной `res`. Для получения числового значения мы используем команду `born=Int32.Parse(res)`. В результате в переменную `born` записывается целое число, введенное пользователем, причем сохраняется это значение именно как число, поэтому с ним можно выполнять арифметические операции.

НА ЗАМЕТКУ

Если пользователь введет в поле диалогового окна не число, то при попытке преобразовать введенный текст в целое число возникнет ошибка. Она также возникает, если пользователь нажмет вместо кнопки **OK** кнопку **Отмена**.

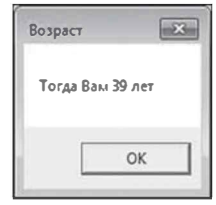


Рис. 1.26. Диалоговое окно с сообщением, содержащим возраст пользователя

Для вычисления возраста пользователя выполняется команда `age=year-born`. Переменной `age`, находящейся слева от оператора присваивания, присваивается значение выражения, находящегося в правой части от оператора присваивания. А в правой части вычисляется разность значений переменных `year` (текущий год) и `born` (год рождения). После вычисления значения переменной `age` она используется в команде `txt="Тогда Вам "+age+" лет"`, которой определяется текстовое значение, запоминаемое с помощью переменной `txt`. В правой части от оператора присваивания указано выражение `"Тогда Вам "+age+" лет"`. Значением выражения является текст, получающийся объединением текстового фрагмента `"Тогда Вам "`, значения переменной `age` и текста `" лет"`. Наконец, командой `MessageBox.Show(txt,"Возраст")` отображается диалоговое окно с названием **Возраст**, в котором есть сообщение с текстом, определяемым переменной `txt`. Как может выглядеть такое окно, показано на рис. 1.26.

После закрытия диалогового окна выполнение программы завершается.

Форматированный вывод

> — В таком виде я не могу. Мне нужно сначала принять ванну, выпить чашечку кофе...
> — Будет тебе там и ванна, будет и кофе, будет и какава с чаем. Поехали!
>
> *из к/ф «Бриллиантовая рука»*

В завершении главы сделаем небольшой комментарий относительно использования методов `WriteLine()` и `Write()` и `Write()`. Мы уже знаем, что методы предназначены для отображения в консольном окне значения, переданного аргументом методу.

НА ЗАМЕТКУ

Напомним, что разница между методами `WriteLine()` и `Write()` состоит в том, что при использовании метода `WriteLine()` после отображения значения в консольном окне выполняется автоматический переход к новой строке. При использовании метода `Write()` такой переход не выполняется.

Но аргументы методу `Write()` или `WriteLine()` можно передавать и несколько иначе. Первым аргументом передается текстовая строка. Эта строка содержит специальные блоки инструкций, которые будем называть *блоками форматирования*. В самом простом случае блок форматирования представляет собой пару фигурных скобок с целочисленным индексом внутри: например `{0}`, `{1}`, `{2}` и так далее. После текстовой строки с блоками форматирования через запятую указываются значения, которые должны быть вставлены вместо блоков форматирования при отображении строки в консоли. Допустим, мы используем команду вида `WriteLine("Значения {0}, {1} и {2}", A, B, C)`. В таком случае в консольном окне отображается текст `"Значения {0}, {1} и {2}"`, но только вместо блока `{0}` вставляется значение переменной `A`, вместо блока `{1}` вставляется значение переменной `B`, а вместо блока `{2}` будет вставлено значение переменной `C`. Фактически индекс в фигурных скобках — это индекс аргумента из списка аргументов, указанных после текстовой строки (первый по порядку элемент после текстовой строки имеет нулевой индекс). В текстовой строке ссылку на один и тот же аргумент можно выполнять несколько раз.

НА ЗАМЕТКУ

Допускается, чтобы переменные или выражения, указанные после первого текстового аргумента в методе `WriteLine()` или `Write()`, были разного типа.

Что касается собственно блоков форматирования, то они могут содержать больше, чем просто индекс аргумента для вставки. Кроме индекса аргумента, в фигурных скобках можно указать способ выравнивания содержимого и формат отображения значения. Общий шаблон для блока форматирования выглядит так (жирным шрифтом выделены ключевые элемента блока форматирования):

{индекс, ширина: формат}

Сначала в фигурных скобках указывается индекс аргумента, который подставляется в соответствующее место при отображении строки. Через запятую после индекса указывается целое число, определяющее ширину поля, выделяемую для отображения значения аргумента. Положительное число означает выравнивание содержимого по правому краю, а отрицательное число означает выравнивание содержимого по левому краю. Также через двоеточие можно указать инструкцию,

определяющую способ (формат) отображения значения аргумента. Для определения формата отображения числового значения используют символ #, обозначающий цифру. Символ X используют как индикатор для отображения числа в шестнадцатеричном формате, символ E является индикатором для использования экспоненциального формата, символ N используют как индикатор десятичного числа, символ C позволяет применять денежный формат. Например, инструкция {0,20:#.##} означает, что при отображении первого аргумента (аргумента с нулевым индексом) следует использовать поле шириной в 20 символов (не меньше) и выравниваться значение будет по правому краю (поскольку ширина поля задана положительным числом). Код #.## означает, что в дробной части числа будет отображаться не больше двух цифр. Если мы хотим, чтобы параметры отображения были те же, но выравнивание выполнялось по левому полю, то используем код {0, -20:#.##} (ширина поля определяется отрицательным числом). Блок {0:E} означает, что число отображается в экспоненциальном формате, а для использования шестнадцатеричного формата отображения числа используем блок {0:X}. Далее в книге по мере необходимости мы еще будем возвращаться к этому вопросу и комментировать способ передачи аргументов методам Write() и WriteLine().

Резюме

> Нет денег. И деньги, и документы, и валюта — все осталось у экскурсовода. Ну, так получилось. Отошли на секундочку, и затерялись в песках.
>
> *из к/ф «Кин-дза-дза»*

- Программа на языке C# обязательно содержит класс с главным методом. Выполнение программы — это выполнение программного кода главного метода.
- Описание класса начинается с ключевого слова class, после которого указывается имя класса и, в фигурных скобках, описывается тело класса — в данном случае тело класса содержит описание главного метода.
- Описание главного метода начинается с ключевого слова static, далее указывают название метода Main() с пустыми круглыми

скобками. Команды главного метода размещаются в блоке из фигурных скобок.

- При необходимости с помощью инструкции `using` подключаются пространства имен. Часто используется пространство имен `System`.
- Для консольного вывода используются статические методы `Write()` и `WriteLine()` класса `Console` из пространства имен `System`. Для отображения диалогового окна используют статический метод `Show()` класса `MessageBox` (доступен после подключения пространства имен `System.Windows.Forms`).
- Для консольного считывания текста используют метод `ReadLine()` из класса `Console`. Текст также можно вводить через окно с полем ввода. Окно отображается статическим методом `InputBox()` из класса `Interaction`. Класс доступен после подключения пространства имен `Microsoft.VisualBasic`.
- Для работы с текстом используют переменные типа `string` (псевдоним для выражения `System.String`). Целые числа реализуются с помощью переменных типа `int`.
- Для преобразования текстового представления числа в целое число используют статический метод `Parse()` из структуры `Int32` (пространство имен `System`).

Задания для самостоятельной работы

> Как говорит наш дорогой шеф, в нашем деле главное — этот самый реализм!
>
> *из к/ф «Бриллиантовая рука»*

1. Напишите программу, в которой пользователь вводит сначала имя, а затем фамилию. Программа выводит сообщение с информацией об имени и фамилии пользователя. Предложите версию программы, в которой ввод и вывод текста осуществляется с помощью диалоговых окон. Также предложите консольную версию программы.

2. Напишите программу, в которой пользователь вводит имя и возраст. Программа отображает сообщение об имени и возрасте пользователя. Предложите консольную версию программы и версию, в которой данные вводятся и выводятся с помощью диалоговых окон.

3. Напишите программу, в которой пользователь последовательно вводит название текущего дня недели, название месяца и дату (номер дня в месяце). Программа выводит сообщение о сегодняшней дате (день недели, дата, месяц). Используйте консольный ввод и вывод данных. Предложите версию программы, в которой для ввода и вывода данных используются диалоговые окна.

4. Напишите программу, в которой пользователю предлагается ввести название месяца и количество дней в этом месяце. Программа выводит сообщение о том, что соответствующий месяц содержит указанное количество дней. Предложите версии программы для ввода/вывода данных через консоль и с помощью диалоговых окон.

5. Напишите программу, в которой по году рождения определяется возраст пользователя. Используйте консольный ввод и вывод данных.

6. Напишите программу, в которой пользователь водит имя и год рождения, а программа отображает сообщение, содержащее имя пользователя и его возраст. Предложите консольную версию программы, а также версию программы, в которой ввод и вывод данных выполняется с помощью диалоговых окон.

7. Напишите программу, в которой по возрасту определяется год рождения. Возраст пользователь вводит в окно с полем, а вычисленный год рождения отображается в другом диалоговом окне. Предложите вариант программы, в которой используется консольный ввод и вывод данных.

8. Напишите программу для вычисления суммы двух чисел. Оба числа вводятся пользователем. Для вычисления суммы используйте оператор +. Предложите два варианта программы: программу, в которой данные вводятся и выводятся с помощью диалоговых окон, и программу, в которой используется консольный ввод и вывод данных.

9. Напишите программу, в которой пользователь вводит число, а программой отображается последовательность из трех чисел: число, на единицу меньшее введенного, введенное число и число, на единицу большее введенного. Предложите версию программы с консольным вводом и выводом данных, а также версию программы, в которой ввод и вывод выполняется с помощью диалоговых окон.

10. Напишите программу, в которой пользователь вводит два числа, а программой вычисляется и отображается сумма и разность этих чисел. Предложите варианты программы с использованием консольного ввода/вывода данных и ввода и вывода данных с помощью диалоговых окон.

Глава 2
БАЗОВЫЕ ТИПЫ И ОПЕРАТОРЫ

> Я представитель цивилизованной планеты и требую, чтобы вы проследили бы за своим лексиконом!
>
> *из к/ф «Кин-дза-дза»*

В этой главе мы рассмотрим базовые типы данных и основные операторы, используемые в языке C#. В частности:

- мы узнаем, какие базовые (или примитивные) типы данных используются в C#;
- разберемся с особенностями объявления переменных;
- выясним, каким образом в языке C# реализуются литералы;
- познакомимся с автоматическим преобразованием типов и узнаем, как выполняется явное приведение типов;
- узнаем, какие базовые операторы есть в языке C#;
- составим представление о способе двоичного кодирования чисел;
- выясним особенности оператора присваивания;
- познакомимся с тернарным оператором.

И это не самый полный перечень тем, которые освещаются в главе. Начнем с базовых типов данных, используемых в языке C#.

Переменные и базовые типы данных

> Милый, это ж театр! Это тебе не в глазок смотреть. Тут в оба надо!
>
> *из к/ф «О бедном гусаре замолвите слово»*

С переменными мы кратко познакомились в предыдущей главе. Напомним, что переменной называется именованный блок памяти. Этот блок может хранить определенное значение. Мы в программе имеем возможность прочитать это значение и изменить его. Для получения доступа к блоку памяти используется переменная. Перед тем как переменную использовать, ее необходимо объявить. Объявление переменной подразумевает, что указывается имя переменной и ее тип. При выполнении команды, которой объявляется переменная, для этой переменной в памяти выделяется место. Каждый раз, когда мы будем обращаться к этой переменной, в действительности будет выполняться обращение к соответствующему месту в памяти. При этом совершенно не важно, где именно в памяти выделено место под переменную. Нам достаточно знать имя переменной, чтобы прочитать ее текущее значение или присвоить новое.

Тип переменной определяет, какие значения могут присваиваться переменной. Также тип переменной влияет на то, какие операции можно выполнять с переменной. В языке C# для каждой переменной должен быть указан тип. После объявления тип переменной не может быть изменен.

Тип переменной определяется с помощью специального идентификатора. Проще говоря, для каждого типа данных имеется свой идентификатор, который указывают при объявлении переменной. Например, мы уже знаем, что если переменная объявлена с идентификатором типа `int`, то значением такой переменной может быть целое число. Кроме целых чисел существуют другие типы данных. С ними мы и познакомимся.

НА ЗАМЕТКУ

Мы познакомимся с базовыми типами данных (иногда их еще называют простыми или примитивными типами). Способы реализации переменных, описываемые здесь, имеют отношение к переменным базовых типов. Помимо базовых типов, существуют еще и ссылочные типы данных. Данные ссылочного типа реализуются несколько иначе по сравнению реализацией данных базовых типов. Ссылочные типы мы рассмотрим при знакомстве с классами и массивами.

В языке C# большинство базовых типов предназначено для реализации числовых значений (здесь имеются в виду целые числа и числа с плавающей точкой), символьных значений (значением символьной переменной может быть отдельный символ) и логических значений (переменная логического типа может принимать одно из двух логических значений — *истина* или *ложь*). Наибольшую группу составляют целочисленные типы — то есть типы данных, предназначенных для реализации целых чисел. Мы уже знакомы с одним из целочисленных типов (речь о типе `int`). Но кроме типа `int` в языке C# существуют и другие целочисленные типы. На первый взгляд может показаться странным, что для реализации целых чисел используется несколько типов данных. Но странного здесь ничего нет: хотя каждый раз имеем дело с целыми числами, записываются они по-разному. Если кратко, то есть две «позиции», по которым отличаются способы реализации целых чисел (при использовании разных типов):

- объем памяти, выделяемой для хранения числа;
- возможность или невозможность использовать отрицательные числа.

ПОДРОБНОСТИ

Для записи целых чисел в памяти компьютера используется битовое представление. Вся память, выделяемая для записи числа, разбита на отдельные блоки — биты. Каждый бит может содержать в качестве значения 0 или 1. Блок из 8 битов называется байтом. Для хранения данных определенного типа выделяется одно и то же количество битов. Например, для записи значений типа `int` в языке C# выделяется 32 бита, или 4 байта. Таким образом, значение типа `int` представляет собой последовательность из 32 нулей и единиц. Такой код интерпретируется как двоичное представление числа. Чем больше выделяется битов для записи числа, тем больше чисел можно закодировать. Если через n обозначить количество битов, с помощью которых кодируются числа, то всего с помощью такого количества битов можно реализовать 2^n разных чисел. Для значений типа `int` количество чисел, которые можно реализовать через переменную данного типа, равно 2^{32}. Диапазон возможных значений для переменной типа `int` ограничен числами -2^{31} (число `-2147483648`) и $2^{31} - 1$ (число `2147483647`) включительно. Всего (с учетом нуля) получается 2^{32} разных чисел.

Поскольку в общем случае приходится кодировать не только положительные, но и отрицательные числа, то один из битов (старший бит) используется как индикатор того, положительное число

или отрицательное. У положительных чисел знаковый бит нулевой, а у отрицательных чисел — единичный. Более детально способы кодирования целых чисел рассматриваются в разделе, посвященном побитовым операторам.

Для реализации целых чисел, кроме типа `int`, используются типы `byte`, `sbyte`, `short`, `ushort`, `uint`, `long` и `ulong`. Значения типов `byte` и `sbyte` кодируются с помощью 8 битов. Различие между типами в том, что значение типа `byte` — это неотрицательное число. Значение типа `sbyte` может быть и положительным, и отрицательным.

НА ЗАМЕТКУ

Начальная буква `s` в названии типа `sbyte` — это «остаток» от слова «signed» (что означает «со знаком»).

С помощью 8 битов можно записать $2^8 = 256$ разных чисел. Поэтому возможное значение для переменной типа `byte` лежит в пределах от `0` до `255` включительно. Значение переменной типа `sbyte` находится в пределах от `-128` до `127` включительно.

Для хранения значения типов `short` и `ushort` выделяется 16 битов. То есть «мощность» этих типов одинакова. Но тип `short` используется для работы с отрицательными и положительными числами, а тип `ushort` используют при работе с неотрицательными числами.

НА ЗАМЕТКУ

Начальная буква `u` в названии типа `ushort`, а также в названиях рассматриваемых далее типов `uint` и `ulong` появилась из слова «unsigned» (что означает «без знака»).

Следующая пара типов `int` (числа со знаком) и `uint` (неотрицательные числа) использует 32 бита для хранения числовых значений. Наконец, самые «мощные» целочисленные типы — это `long` и `ulong`. Переменные данных типов хранятся в ячейках памяти объемом в 64 бита. Значения типа `long` интерпретируются как числа со знаком, а значения типа `ulong` обрабатываются как неотрицательные числа. Более подробная информация не только о целочисленных типах, но и о прочих базовых типах языка C# приведена в табл. 2.1.

Табл. 2.1. Базовые типы языка C#

Тип	Память в битах	Диапазон значений	Описание	Структура
`byte`	8	от 0 до 255	Целое неотрицательное число	`Byte`
`sbyte`	8	от –128 до 127	Целое число	`SByte`
`short`	16	от –32768 до 32767	Целое число	`Int16`
`ushort`	16	от 0 до 65535	Целое неотрицательное число	`UInt16`
`int`	32	от –2147483648 до 2147483647	Целое число	`Int32`
`uint`	32	от 0 до 4294967295	Целое неотрицательное число	`UInt32`
`long`	64	от –9223372036854775808 до 9223372036854775807	Целое число	`Int64`
`ulong`	64	от 0 до 18446744073709551615	Целое неотрицательное число	`UInt64`
`float`	32	от 1.5E-45 до 3.4E+38	Действительное число (с плавающей точкой)	`Single`
`double`	64	от 5E-324 до 1.7E+308	Действительное число (с плавающей точкой) двойной точности	`Double`
`decimal`	128	от 1E-28 до 7.9E+28	Действительное число для выполнения особо точных (финансовых) расчетов	`Decimal`
`char`	16	от 0 до 65535	Символьное значение	`Char`
`bool`	8	значения `true` и `false`	Логическое значение	`Boolean`

Для удобства в табл. 2.1 кроме идентификаторов для базовых типов указано количество битов, выделяемых для значения соответствующего типа, а также приведен диапазон возможных значений для переменной соответствующего типа.

НА ЗАМЕТКУ

В действительности значения базовых типов реализуются как экземпляры структур. Структуры мы рассмотрим немного позже. В табл. 2.1 для каждого базового типа указана структура, которая соответствует данному типу. Фактически идентификаторы базовых

типов являются псевдонимами для названий структур. Например, для работы с целыми числами можно создать переменную базового типа `int`, что означает создание экземпляра структуры `Int32`. То есть ситуация не самая тривиальная, но в большинстве случаев это не играет никакой роли.

Кроме целых чисел часто приходится иметь дело с действительными числами, или числами с плавающей точкой. Такие числа имеют целую часть и дробную, которая указывается через точку после целой части.

НА ЗАМЕТКУ

Для чисел с плавающей точкой также можно использовать экспоненциальное (компьютерное) представление в виде мантиссы и показателя степени. Действительное число представляется в виде произведения мантиссы (действительное число) на 10 в целочисленной степени. Записать число в таком представлении можно следующим образом: указать мантиссу, затем символ `E` (или `e`) и целое число, определяющее показатель степени. Например, выражение `1.5E-45` означает число $1{,}5 \times 10^{-45}$, а выражение `3.4E+38` соответствует числу $3{,}4 \times 10^{38}$. Если показатель степени положительный, то знак `+` можно не указывать.

Для реализации действительных чисел используются типы `float`, `double` и `decimal`. Для значения типа `float` выделяется 32 бита, а под значение типа `double` выделяется 64 бита. Поэтому числа, реализованные как значения типа `double`, имеют большую «точность». Наименьшее по модулю `float`-число равно $1{,}5 \times 10^{-45}$, а самое маленькое по модулю `double`-число равно 5×10^{-324}. Верхняя граница для значений типа `double` составляет величину $1{,}7 \times 10^{308}$, а `float`-значения ограничены сверху числом $3{,}4 \times 10^{38}$. В этом смысле тип `double` является более предпочтительным.

НА ЗАМЕТКУ

Есть и другая причина для приоритетного использования типа `double` по сравнению с типом `float`. Ее мы обсудим немного позже.

Помимо типов `float` и `double` в языке C# есть тип `decimal`. Под значение типа `decimal` выделяется 128 битов, что в два раза больше, чем для типа `double`. Но главное назначение такой мощной

«поддержки» — обеспечить высокую точность вычислений, выполняемых со значениями типа `decimal`. Обычно необходимость в высокой точности операций возникает при финансовых расчетах. Поэтому обычно тип `decimal` упоминают как специальный тип, предназначенный для поддержки высокоточных финансовых вычислений.

Для работы с символьными значениями предназначен тип `char`. Значением переменной типа `char` может быть отдельный символ (буква). Для хранения такого значения в памяти выделяется 16 битов.

ПОДРОБНОСТИ

Значения типа `char` кодируются так же, как и целые числа — имеем дело с 16 битами, заполненными нулями и единицами. Аналогичным образом кодируются, например, значения типа `ushort`. Поэтому технически `char`-значение можно рассматривать как неотрицательное числовое значение, которое попадает в диапазон целых чисел от `0` до `65535`. Но обрабатывается такое значение несколько иначе (по сравнению с обработкой целочисленных значений), а именно: по двоичному коду определяется число, и затем из кодовой таблицы берется символ с соответствующим кодом.

Наконец, тип `bool` предназначен для работы с логическими значениями. Поэтому этот тип обычно называют логическим. Переменная логического типа может принимать одно из двух значений: `true` (истина) или `false` (ложь). Значения логического типа используются в управляющих инструкциях — таких, как условный оператор или операторы цикла.

Литералы

— Что они хотят?
— Ку они хотят...
из к/ф «Кин-дза-дза»

Под литералами подразумевают константные (такие, которые нельзя изменить) значения, используемые в программном коде. Примерами литералов являются число `123` (целочисленный литерал), текст `"Изучаем C#"` (текстовый литерал) или буква `'ы'` (символьный

литерал). То есть это некоторые фиксированные значения, понятные для программиста, которые могут использоваться в программе. Но пикантность ситуации в том, что такие значения в программе реализуются по тому же принципу, что и переменные. Другими словами, каждый литерал имеет определенный тип (или относится к определенному типу), и обрабатывается литерал в соответствии с правилами работы со значением данного типа. Возникает вопрос: к какому типу относится, например, целочисленный литерал `123`? Понятно, что это целое число. Но для реализации целочисленных значений существует несколько типов, и в данном конкретном случае любой из них теоретически подходит для реализации числа `123`. Ответ же состоит в том, что число `123` реализуется как `int`-значение. Вообще есть несколько правил, определяющих способ реализации литералов.

- Целочисленный литерал реализуется как значение «наименьшего» типа, начиная с типа с `int`, достаточного для сохранения значения литерала.
- Действительные числовые литералы (число с плавающей точкой, как, например, число `12.3`) реализуются в виде значений типа `double`.
- Символьные литералы заключаются в одинарные кавычки (например, `'F'` или `'я'`) и реализуются как значения типа `char`.
- Текстовые литералы (текст) заключаются в двойные кавычки (например, `"Язык C#"` или `"A"`) и реализуются как объекты класса `String` из пространства имен `System` (мы используем идентификатор `string` для выражения `System.String`).

НА ЗАМЕТКУ

Символьный литерал заключается в одинарные кавычки, а текстовый — в двойные. Символьный литерал — это один символ. Текст может содержать много символов. Но если один символ заключить в двойные кавычки, то это будет текст, а не символьный литерал. Иными словами, выражение `'A'` представляет собой символьный литерал, а выражение `"A"` является текстовым литералом (состоящим из одного символа). И хотя с формальной точки зрения в обоих случаях мы имеем дело с одним и тем же символом, с технической точки зрения мы имеем дело со значениями разных типов, реализованных совершенно по-разному.

Описанные выше правила применяются при реализации литералов по умолчанию. Но мы можем вносить в этот процесс разумные коррективы. Так, если мы хотим, чтобы целочисленный литерал реализовался как `long`-значение, надо использовать суффикс `L` или `l`. Например, литерал `123L` означает число `123`, реализованное как значение типа `long`. Если воспользоваться суффиксом `U` или `u`, то целочисленный литерал будет реализован как значение типа `uint`. Суффикс `UL` (а также `ul`, `Ul` и `uL`) позволяет определить литерал типа `ulong` — например, выражение `123UL` определяет число `123`, реализованное как значение типа `ulong`.

Если необходимо, чтобы числовой литерал реализовался как `float`-значение, используют суффикс `F` или `f`. Например, выражение `12.3F` определяет число `12.3` как значение типа `float`.

Если использовать суффикс `M` или `m` в числовом литерале, получим число, реализованное как значение типа `decimal`. Примером может быть выражение `12.3M`.

НА ЗАМЕТКУ

Целочисленные значения можно представлять шестнадцатеричными литералами. Шестнадцатеричный литерал начинается с символов `0x` или `0X`. В представлении шестнадцатеричного литерала используются цифры от `0` до `9`, а также латинские буквы от `A` до `F` для обозначения чисел от `10` до `15` соответственно. Буквы можно использовать как строчные, так и прописные. Если шестнадцатеричный литерал записан в виде последовательности цифр и букв $\overline{a_n a_{n-1} \ldots a_2 a_1 a_0}$, в которой a_k (k=0, 1, 2, ..., n) — это цифры от 0 до 9 или буквы от A до F, то в десятичной системе счисления такое число вычисляется как $\overline{a_n a_{n-1} \ldots a_2 a_1 a_0} = a_0 \times 16^0 + a_1 \times 16^1 + a_2 \times 16^2 + \ldots + a_{n-1} \times 16^{n-1} + a_n \times 16^n$. При этом вместо букв A, B... F подставляются числа 10,11...15. Например, литералу `0x2F9D` соответствует десятичное число $13 \times 16^0 + 9 \times 16^1 + 15 \times 16^2 + 2 \times 16^3 = 12189$.

Управляющие символы

— И быть тебе за это рыбой, мерзкой и скользкой!
— Да, но обещали котом!
— Недостоин!

из к/ф «Формула любви»

В C# есть группа символов, которые называются *управляющими*. Формально это символы, но при их «печатании» обычно выполняется определенное действие. Все управляющие символы начинаются с обратной косой черты \. После обратной косой черты указывается некоторый символ. Например, выражение `\n` является инструкцией перехода к новой строке, а инструкция `\t` представляет собой инструкцию для выполнения горизонтальной табуляции.

> **НА ЗАМЕТКУ**
>
> Хотя выражение `\n` или `\t` формально состоит из двух символов, интерпретируется каждое из них как один символ. Поэтому такое выражение можно присвоить значением символьной переменной: выражение как символ заключается в одинарные кавычки (получаем литералы `'\n'` и `'\t'`), и такой литерал присваивается значением переменной типа `char` (или используется другим образом).

Если в текстовом литерале присутствует инструкция `\n` и такой литерал передан аргументом методу `WriteLine()`, то при отображении этого литерала в соответствующем месте будет выполнен переход к новой строке. Инструкция `\t` в процессе отображения литерала обрабатывается так: курсор оказывается в ближайшей позиции табуляции.

> **НА ЗАМЕТКУ**
>
> Строку, в которую выводится текст, можно условно разделить на «позиции», предназначенные для отображения символов. Среди этих «позиций» есть «особые» или «реперные» — обычно это каждая восьмая «позиция». То есть «особыми» являются 1-я, 9-я, 17-я, 25-я и так далее позиции. Когда нужно «напечатать» инструкцию `\t`, то курсор «прыгает» в ближайшую «реперную» позицию. Следующий символ будет отображаться именно там.

Кроме перечисленных выше, можно выделить управляющие символы \a (звуковой сигнал), \b (инструкция перемещения курсора на одну позицию назад) и \r (инструкция перемещения курсора в начало текущей строки).

ПОДРОБНОСТИ

Нередко возникает потребность использовать в текстовых и символьных литералах двойные и одинарные кавычки или обратную косую черту. Проблема в том, что просто так добавить соответствующий символ не получится: обратная косая черта является признаком управляющего символа, а двойные и одинарные кавычки используются для выделения литералов (текстовых и символьных соответственно). Решение проблемы состоит в том, чтобы использовать дополнительную обратную косую черту. Например, двойные кавычки в литерал добавляются в виде выражения \", выражение \' можно использовать для добавления в литерал одинарной кавычки, а выражение \\ полезно, если необходимо добавить в литерал (одну) обратную косую черту.

НА ЗАМЕТКУ

Если перед текстовым литералом указать символ @, то такой литерал становится «буквальным» или «не форматируемым»: он отображается так, как записан. Такой литерал не подлежит форматированию, он используется «как есть». Например, если мы передадим аргументом методу `WriteLine()` текстовый литерал "Изучаем\n язык C#", то из-за наличия в литерале инструкции \n в первой строке консольного окна появится текст "Изучаем", а во второй — текст " язык C#". Но если передать аргументом буквальный текстовый литерал @"Изучаем\n язык C#", то в консольном окне появится текст "Изучаем\n язык C#".

Преобразование типов

— Может, его за Нерона нам выдать — как будто?
— Где же ты видел Неронов в лаптях и онучах?

из к/ф «О бедном гусаре замолвите слово»

Нередко в выражения входят значения разных типов. Скажем, может возникнуть необходимость к значению типа `float` прибавить значение

типа `int` и полученный результат присвоить переменной типа `double`. Математически здесь все корректно, поскольку речь идет об арифметической операции с числами, результат этой операции представляется в виде действительного числа, а целые числа являются подмножеством чисел действительных. Но в плане программной реализации подобной операции мы имеем дело со значениями трех разных типов. К счастью, в языке C# есть встроенные механизмы, позволяющие сводить к минимуму неудобства, связанные с использованием в выражениях значений разных типов. В ситуациях, подобных описанной выше, выполняется *автоматическое преобразование типов*. Обычно речь идет о преобразованиях различных числовых типов. Есть несколько правил выполнения таких преобразований. Итак, допустим, имеются два операнда, которые относятся к разным числовым типам. Алгоритм выполнения преобразований такой:

- Если один из операндов относится к типу `decimal`, то другой автоматически преобразуется к типу `decimal`. Но если другой операнд — значение типа `float` или `double`, то возникает ошибка.
- Если один операнд — типа `double`, то другой автоматически преобразуется в тип `double`.
- Если один операнд относится к типу `float`, то другой операнд будет преобразован в значение типа `float`.
- Если в выражении есть операнд типа `ulong`, то другой операнд автоматически приводится к типу `ulong`. Ошибка возникает, если другой операнд относится к типу `sbyte`, `short`, `int` или `long`.
- Если в выражении есть операнд типа `long`, то другой операнд будет преобразован в тип `long`.
- Если один из операндов относится к типу `uint`, а другой операнд — значение типа `sbyte`, `short` или `int`, то оба операнда расширяются до типа `long`.
- Если в выражении оба операнда целочисленные, то они преобразуются в тип `int`.

Эти правила применяются последовательно. Например, если в выражении один из операндов относится к типу `decimal`, то выполняется попытка преобразовать второй операнд к тому же типу, и если попытка успешная, то вычисляется результат. Он также будет относиться к типу `decimal`. До выполнения прочих правил дело не доходит. Но если

в выражении нет операндов типа decimal, тогда в игру вступает второе правило: при наличии операнда типа double другой операнд преобразуется к этому же типу, такого же типа будет результат. Далее, если в выражении нет ни операнда типа decimal, ни операнда типа double, то применяется третье правило (для типа float), и так далее.

В этой схеме стоит обратить внимание на два момента. Во-первых, общий принцип состоит в том, что автоматическое преобразование выполняется к большему (в плане диапазона допустимых значений) типу, так, чтобы потеря значения исключалась — то есть автоматическое преобразование выполняется всегда с расширением типа. Отсюда становится понятно, почему при попытке преобразовать значения типа double и float к значению типа decimal возникает ошибка. Хотя для значений типа decimal выделяется 128 битов (и это больше, чем 64 бита для типа double и 32 бита для типа float), но диапазон допустимых значений для типа decimal меньше, чем диапазоны допустимых значений для типов double и float. Поэтому теоретически преобразуемое значение может быть потеряно, а значит, такое автоматическое преобразование не допускается.

НА ЗАМЕТКУ

Подчеркнем, что здесь речь идет об автоматическом преобразовании типов — то есть о преобразовании, которое в случае необходимости выполняется автоматически. Помимо автоматического преобразования типов существует еще и явное приведение типа, которые выполняется с помощью специальной инструкции. Там правила другие. Сама инструкция явного приведения типа имеет вид (тип) выражения: идентификатор типа, к которому следует привести значение выражения, указывается в круглых скобках перед этим выражением.

Во-вторых, если в выражении оба операнда целочисленные, то даже если ни один из них не относится к типу int (например, два операнда типа short), то каждый из операндов приводится к типу int, и, соответственно, результат также будет типа int. Это простое обстоятельство имеет серьезные последствия. Допустим, мы командой short a=10 объявили переменную a типа short со значением 10. Если после этого мы попытаемся использовать команду a=a+1, то такая команда будет некорректной и программный код не скомпилируется. Причина в том, что числовой литерал 1, использованный в команде, относится к типу int. Поэтому в выражении a+1 текущее значение переменной

a преобразуется к типу int, и результат выражения a+1 также вычисляется как int-значение. Таким образом, получается, что мы пытаемся присвоить значение int-типа переменной типа short. А это запрещено, поскольку может произойти потеря значения.

ПОДРОБНОСТИ

При вычислении выражения a+1 считывается значение переменной a, и это число преобразуется к типу int. Сама переменная как была типа short, так переменной типа short и остается.

Решение проблемы может состоять в том, чтобы использовать явное приведение типа. Корректная команда выглядит так: a=(short)(a+1). Здесь идентификатор short в круглых скобках перед выражением (a+1) означает, что после вычисления значения выражения полученный результат (значение типа int) следует преобразовать в значение типа short.

ПОДРОБНОСТИ

Для значения типа int в памяти выделяется 32 бита. Значение типа short запоминается с помощью 16 битов. Преобразование значения типа int в значение типа short выполняется путем отбрасывания содержимого в старших 16 битах int-значения. Если int-число не очень большое (не выходит за диапазон short-значения), то старшие биты заполнены нулями (для положительного числа), и при их отбрасывании значение потеряно не будет. Но даже если в старших битах не нули, то биты все равно отбрасываются, и значение может быть потеряно. Поэтому преобразование значения типа int в значение типа short — это потенциально опасное преобразование. Такие преобразования автоматически не выполняются. В некотором смысле инструкция (short) свидетельствует о том, что программист понимает риск и берет ответственность на себя.

Есть еще одно важное обстоятельство: при инициализации переменной short командой short a=10 переменной типа short присваивается литерал 10, который является значением типа int. Хотя формально здесь тоже вроде бы противоречие, но при присваивании целочисленной переменной целочисленного литерала, если последний не выходит за допустимые пределы для переменной данного типа, приведение типов выполняется автоматически.

Еще один пример: командой byte a=1, b=2, c объявляем три переменные (a, b и c) типа byte. Переменные a и b инициализируются

со значениями 1 и 2 соответственно. Но если мы захотим воспользоваться командой c=a+b, то такая команда также будет некорректной. В отличие от предыдущего случая здесь нет литерала типа int. Здесь вычисляется сумма a+b двух переменных типа byte, и результат присваивается (точнее, выполняется попытка присвоить) переменной типа byte. Проблема же в том, что, хотя в выражении a+b оба операнда относятся к типу byte, они автоматически преобразуются к типу int (в соответствии с последним пунктом в списке правил автоматического преобразования типов), и поэтому результатом выражения a+b является значение типа int. Получается, что мы пытаемся присвоить byte-переменной int-значение, а это неправильно. Чтобы исправить ситуацию, необходимо использовать явное приведение типа. Корректная версия команды присваивания выглядит как c=(byte)(a+b).

НА ЗАМЕТКУ

Если преобразование целых чисел в действительные в случае необходимости выполняется автоматически, то обратное преобразование требуется выполнять в явном виде. Например, если нам нужно преобразовать значение типа double в значение типа int, то перед соответствующим выражением указывается инструкция (int). Преобразование выполняется отбрасыванием дробной части действительного числа. Например, если командой double x=12.6 объявляется переменная x типа double со значением 12.6, а командой int y объявляется переменная y типа int, то в результате выполнения команды y=(int)x переменная y получит значение 12. При этом значение переменной x не меняется.

Арифметическое выражение может содержать не только числовые, но и символьные операнды. В таких случаях char-операнд автоматически преобразуется к int-типу. Например, выражение 'A'+'B' имеет смысл. Результат такого выражения — целое число 131. Объяснение простое: значения типа char (то есть символы) хранятся в виде целых чисел. Это последовательность из 16 битов. Вопрос только в том, как ее интерпретировать. Для целых чисел такая последовательность интерпретируется как двоичное представление числа. Для символов «преобразование» двоичного кода выполняется в два этапа: сначала по двоичному коду определяется число, а затем по этому числу определяется символ в кодовой таблице. Проще говоря, 16 битов для char-значения — это код символа в кодовой таблице. Если символ нужно преобразовать в целое число, то вместо символа используется его код из кодовой

таблицы. У буквы 'A' код 65, а у буквы 'B' код 66, поэтому сумма символов 'A'+'B' дает целочисленное значение 131.

Преобразование типа char в тип int в случае необходимости выполняется автоматически. Обратное преобразование (из целого числа в тип char) выполняется в явном виде. Например, результатом команды char s=(char)65 является объявление символьной переменной s со значением 'A'. Принцип преобразования целого числа в символ такой: преобразуемое целочисленное значение определяет код символа в кодовой таблице. Так, поскольку у буквы 'A' код 65, то значением выражения (char)65 будет символ 'A'.

НА ЗАМЕТКУ

Учитывая правила автоматического преобразования типов и способ реализации числовых литералов, можно дать следующую рекомендацию: если нет объективной необходимости поступать иначе, то для работы с целочисленными значениями разумно использовать переменные типа int, а для работы с действительными значениями имеет смысл использовать переменные типа double.

Объявление переменных

> Огонь тоже считался божественным, пока Прометей не выкрал его. Тенерь мы кипятим на нем воду.
>
> *из к/ф «Формула любви»*

Мы уже знаем (в общих чертах), что такое переменная и как она определяется. Здесь сделаем некоторое обобщение. Итак, напомним, что переменная предназначена для хранения в памяти некоторого значения. Переменная предоставляет пользователю доступ к области памяти, где хранится значение. С помощью переменной это значение можно прочитать и, в случае необходимости, изменить. Значение переменной присваивается в формате переменная=значение, то есть используется оператор присваивания =, слева от которого указывается переменная, которой присваивается значение, а справа — значение, присваиваемое переменной.

Перед использованием переменная должна быть объявлена. Сделать это можно практически в любом месте программного кода, но обязательно

до того, как переменная начинает использоваться. Объявляется переменная просто: указывается ее тип и название. При этом можно:

- Объявлять сразу несколько переменных, если они относятся к одному типу. В таком случае после идентификатора типа названия переменных указываются через запятую. Например, инструкцией `int A, B, C` объявляются три переменные (`A`, `B` и `C`), каждая типа `int`.
- При объявлении переменной можно сразу присвоить значение (инициализировать переменную). Так, командой `short x=12` объявляется переменная `x` типа `short`, и этой переменной присваивается значение `12`. Командой `int A=5, B, C=10` объявляются переменные `A`, `B` и `C` типа `int`, переменной `A` присваивается значение `5`, а переменной `C` присваивается значение `10`.
- Разрешается выполнять *динамическую инициализацию переменной*: это когда переменная инициализируется на основе выражения, в которое входят другие переменные. Главное условие для динамической инициализации состоит в том, что переменные, входящие в выражение, на момент вычисления выражения должны иметь значения. Проще говоря, для динамической инициализации можно использовать только те переменные, которым уже присвоены значения. Например, сначала с помощью команды `int x=10, y=20` мы можем объявить переменные `x` и `y` со значениями `10` и `20` соответственно, а затем командой `int z=x+y` объявляем и динамически инициализируем переменную `z`. Значение этой переменной будет равно `30`, поскольку на момент вычисления выражения `x+y` переменные `x` и `y` имеют значения `10` и `20`. Важный момент: между переменными `z`, с одной стороны, и `x` и `y`, с другой стороны, функциональная связь не устанавливается. Если после инициализации переменной `z` мы изменим значение переменной `x` или `y`, то значение переменной `z` не изменится.
- При объявлении переменной вместо идентификатора типа можно использовать ключевое слово `var`. С помощью одной `var`-инструкции может объявляться одна переменная (то есть нельзя использовать одну `var`-инструкцию для объявления сразу нескольких переменных). Переменная, объявляемая в `var`-инструкции, должна сразу инициализироваться, и тип такой переменной определяется типом литерала (или типом значения выражения), инициализирующим переменную. Например, командой `var z=10`

объявляется переменная z. Переменная z относится к типу int, поскольку значение 10, которое присваивается переменной, представляет собой целочисленный литерал, а целочисленные литералы реализуются как значения типа int. Но если мы воспользуемся командой var z=10.0, то в таком случае переменная z будет типа double, поскольку литерал 10.0 реализуется как значение типа double. Наконец, команда var z=10F определяет переменную z типа float. Причина проста: литерал 10F (из-за суффикса F) реализуется как float-значение, и это определяет тип переменной z. Ситуация может быть и более сложной. Скажем, командой byte x=1, y=2 объявляется две byte-переменные (x и y со значениями 1 и 2 соответственно). Если после этого мы объявим переменную z командой var z=x+y, то значением переменной z будет число 3, а сама переменная будет типа int. Почему? Потому что при вычислении выражения x+y, несмотря на то что оба операнда являются byte-переменными, по правилам автоматического преобразования типов их значения преобразуются в числа типа int, и результат (число 3) также представлен числом типа int. Поэтому тип переменной z — целочисленный тип int.

- Наряду с переменными могут использоваться и *константы*. Принципиальное отличие константы от переменной состоит в том, что значение константы изменить нельзя. Как и переменная, константа объявляется. Но кроме идентификатора типа и имени константы, указывается еще и ключевое слово const. При объявлении константа инициализируется: указанное при инициализации значение константы впоследствии изменить не получится. Например, командой const int num=123 объявляется константа num типа int, и значение этой константы равно 123. Командой const char first='A', second='B' объявляются две символьные константы first и second со значениями 'A' и 'B' соответственно.

Существует такое понятие, как *область доступности* (или область видимости) переменной. Это место в программном коде, где переменная может использоваться. Правило очень простое: переменная доступна в том блоке, в котором она объявлена. Блок, в свою очередь, определяется парой фигурных скобок. Поэтому если переменная объявлена в главном методе программы, то она доступна в этом методе (начиная с того места, где переменная объявлена).

ПОДРОБНОСТИ

Далее мы узнаем, что главным методом область использования переменных не ограничивается. Есть и другие хорошие места в программе, где могут использоваться переменные. Более того, даже в главном методе могут быть внутренние блоки. Скажем, часть команд в главном методе мы можем заключить в фигурные скобки. Другими словами, блоки кода могут содержать внутренние блоки кода, которые, в свою очередь, также могут содержать внутренние блоки, и так далее. Переменная, объявленная в некотором блоке (например, в главном методе), доступна и во всех внутренних блоках. Но не наоборот: если переменная объявлена во внутреннем блоке, то во внешнем она не доступна. Допустим, главный метод программы имеет такую структуру:

```
static void Main(){
    // Переменная объявлена в главном методе:
    int A;
    // Команды
    {// Начало внутреннего блока
        // Переменная объявлена во внутреннем блоке:
        int B;
        // Команды
    }// Завершение внутреннего блока
    // Команды
}
```

Здесь мы имеем дело с переменной `A`, объявленной в главном методе, и переменной `B`, объявленной во внутреннем блоке. Причем внутренний блок — это часть команд, заключенных в фигурные скобки. В этом случае переменная `A` доступна везде в главном методе, включая внутренний блок. Но вот переменную `B` можно использовать только во внутреннем блоке. Вне этого блока переменная «не существует».

Название переменной, объявленной во внутреннем блоке, не может совпадать с названием переменной, объявленной во внешнем блоке. Это означает, что если мы объявили в главном методе переменную `A`, то объявить во внутреннем блоке главного метода переменную с таким же именем мы не можем.

Арифметические операторы

— Что они кричат?
— Продают.
— Что продают?
— Все нродают.

из к/ф «Кин-дза-дза»

Далее мы рассмотрим основные операторы, используемые в языке C#. Начнем с *арифметических операторов*, используемых при выполнении арифметических операций. Вообще же все операторы в языке C# обычно разделяют на четыре группы:

- арифметические операторы;
- логические операторы;
- операторы сравнения;
- побитовые операторы.

Также в языке C# есть тернарный оператор. Да и оператор присваивания не такой простой, как может показаться на первый взгляд. Все эти вопросы мы обязательно обсудим. Ну а пока — арифметические операторы.

ПОДРОБНОСТИ

Оператор — это некий символ, обозначающий определенную операцию. Оператор используется с операндом или операндами. Операция, определяемая оператором, выполняется с операндами. Если оператор используется с одним операндом, то такой оператор называется унарным. Если оператор используется с двумя операндами, то такой оператор называется бинарным. В языке C# также есть один оператор, который используется аж с тремя операндами. Этот оператор называется тернарным.

Назначение большинства арифметических операторов интуитивно понятно каждому, кто хотя бы в минимальном объеме сталкивался с математикой. В табл. 2.2 представлен полный перечень арифметических операторов языка C#. Там же дано краткое их описание.

Табл. 2.2. Арифметические операторы

Оператор	Описание
`+`	Оператор сложения. Значением выражения вида `A+B` является сумма значений переменных `A` и `B`
`-`	Оператор вычитания. Значением выражения вида `A-B` является разность значений переменных `A` и `B`
`*`	Оператор умножения. Значением выражения вида `A*B` является произведение значений переменных `A` и `B`
`/`	Оператор деления. Значением выражения вида `A/B` является частное значений переменных `A` и `B`. Если оба операнда `A` и `B` — целочисленные, то деление выполняется нацело. Для выполнения обычного деления с целочисленными операндами перед выражением указывают инструкцию `(double)`
`%`	Оператор вычисления остатка от деления. Значением выражения вида `A%B` является остаток от целочисленного деления переменных `A` и `B`
`++`	Оператор инкремента. При выполнении команды вида `A++` или `++A` значение переменной `A` увеличивается на `1`
`--`	Оператор декремента. При выполнении команды вида `A--` или `--A` значение переменной `A` уменьшается на `1`

Описание операторов дано в предположении, что операнд (или операнды) является числовым. Вместе с тем это не всегда так. Так, в операции сложения с использованием оператора + один или оба операнда могут быть, например, текстовыми. В таком случае выполняется объединение текстовых строк. Скажем, если к тексту прибавить число, то числовой операнд автоматически преобразуется в текстовый формат и результатом выражения является текст, получающийся объединением «суммируемых» текстовых фрагментов.

Имеет свои особенности и оператор деления. Дело в том, что если делить два целых числа, то выполняется целочисленное деление. Например, результатом выражения `12/5` будет не `2.4`, как можно было бы ожидать, а целое число `2` (целая часть от деления `12` на `5`). Но вот результат выражения `12.0/5` или `12/5.0` — это число `2.4`. Объяснение простое: в выражении `12/5` оба операнда целочисленные, а в выражениях `12.0/5` и `12/5.0` целочисленный только один операнд. Поэтому в выражении `12/5` речь идет о целочисленном делении, а значения выражений `12.0/5` и `12/5.0` вычисляются с использованием обычного деления. Если `a` и `b` — целочисленные переменные (например, типа `int`), то при вычислении выражения `a/b` используется целочисленное деление. Если нужно, чтобы деление было обычным, используют команду вида `(double)a/b`.

Оператор % позволяет вычислить остаток от целочисленного деления. Причем операнды могут быть не только целыми, но и действительными числами. В таком случае результатом выражения вида A%B может быть не только целое, но и действительное число.

ПОДРОБНОСТИ

Если A и B некоторые числа (действительные, в том числе они могут быть и отрицательными), то остаток от деления A на B (обозначим эту операцию как A%B) вычисляется следующим образом. Сначала находится наибольшее по модулю целое число n, такое, что |nB| ≤ |A| при условии совпадения знаков A и nB, и результат операции A%B вычисляется как A%B = A – nB. Например, если A = 13,7 и B = 3,1, то n = 4 (nB = 12,4) и A%B = A – nB = 13,7 – 12,4 = 1,3. Если A = –13,7 и B = 3,1, то n = –4 (nB = –12,4) и A%B = A – nB = –13,7 + 12,4 = –1,3, и так далее.

Если A и B — целочисленные переменные, то результат выражения A%B совпадает с разностью значений переменной A и выражения B*(A/B) (то есть значения выражений A%B и A-B*(A/B) в этом случае совпадают).

Если все предыдущие операторы (+, –, *, / и %) были бинарными, то операторы инкремента ++ и декремента –– являются унарными — они используются с одним, а не с двумя операндами. У каждого из этих операторов есть *префиксная* и *постфиксная* формы: в префиксной форме операторы инкремента и декремента указываются перед оператором (например, ++A или ––A), а в постфиксной форме сначала указывается операнд, а затем — оператор (например, A++ или A––). В плане влияния на значение операнда префиксная и постфиксная формы эквивалентны. При выполнении команды вида ++A или A++ значение переменной A увеличивается на 1. Команда вида ––A и A–– означает уменьшение значения переменной A на 1. Но разница между префиксными и постфиксными формами все же есть. Дело в том, что выражения вида ++A, A++, ––A и A–– имеют значения. Проще говоря, инструкцию ++A или, скажем, A–– мы можем интерпретировать как некоторое число (если операнд A числовой). То есть, помимо прямых последствий для операнда, само выражение на основе оператора инкремента или декремента может использоваться в более сложном выражении. Правило такое: значением выражения с оператором инкремента или декремента в префиксной форме является новое (измененное) значение операнда, а значением выражения с оператором инкремента или декремента в постфиксной форме является старое

(до изменения) значение операнда. Например, командой `int A=10, B` объявляются две переменные `A` и `B`, и переменная `A` получает значение `10`. Если затем выполняется команда `B=A++`, то переменная `B` получит значение `10`, а значение переменной `A` станет равным `11`. Почему? Потому что при выполнении инструкции `A++` значение переменной `A` увеличивается на `1` и становится равным `11`. Но переменной `B` значением присваивается не значение переменной `A`, а значение выражения `A++`. Значение выражения `A++`, поскольку использована постфиксная форма оператора инкремента, — это старое (исходное) значение переменной `A` (то есть значение `10`). А вот если бы мы вместо команды `B=A++` использовали команду `B=++A`, то и переменная `A`, и переменная `B` получили бы значение `11`. Причина в том, что значением выражения `++A` с оператором инкремента в префиксной форме является новое значение переменной `A` (число `11`).

НА ЗАМЕТКУ

После выполнения команд `int A=10, B` и `B=A--` у переменной `A` будет значение `9`, а у переменной `B` будет значение `10`. Если команду `B=A--` заменить на `B=--A`, то обе переменные `A` и `B` будут иметь значение `9`.

При выполнении арифметических операций, помимо числовых операндов, могут использоваться и символьные значения (значения типа `char`). В таком случае выполняется автоматическое преобразование значения типа `char` в числовое значение (используется код соответствующего символа из кодовой таблицы). В этом смысле мы вполне можем вычислить, например, разность `'D'-'A'` (результат равен `3` — разность кодов символов `'D'` и `'A'`).

ПОДРОБНОСТИ

Операторы инкремента и декремента также могут использоваться с символьным операндом. Но здесь есть одна особенность. Допустим, командой `char symb='A'` мы объявили символьную переменную `symb` со значением `'A'`. Если мы используем выражение `symb+1`, то его значением является целое число `66` (к коду `65` символа `'A'` прибавляется `1`). Поэтому, чтобы с помощью такого выражения в переменную `symb` записать следующий символ после символа `'A'`, нужно использовать команду `symb=(char)(symb+1)`. Буквы алфавита в кодовой таблице идут одна за другой, поэтому следующим после символа `'A'` будет символ `'B'`, затем символ `'C'` и так далее.

Значение выражения `symb+1` — число `66`, которое является кодом символа `'B'` в кодовой таблице. Результат выражения `symb+1` явно приводится к символьному виду, и полученное число интерпретируется как код символа. В результате значением переменной `symb` будет символ `'B'`.

А вот если мы воспользуемся выражением `symb++` (или `++symb`), то значение переменной `symb` станет равным `'B'`. Дело в том, что в плане последствий для операнда `A` выражение `A++` (или `++A`) эквивалентно команде вида `A=(тип A)(A+1)`. То есть к исходному значению операнда `A` прибавляется единица, полученный результат приводится к типу операнда `A`, и это значение присваивается операнду `A`. Если операнд `A` — числовой, то факт приведения типа не столь важен. Но вот если операнд символьный — то момент с приведением типа важен. Это же замечание относится и к оператору декремента.

НА ЗАМЕТКУ

Помимо бинарных операторов сложения `+` и вычитания `-` есть еще унарные операторы «плюс» `+` и «минус» `-`. Унарный «минус» пишется перед отрицательными числами (то есть используется для обозначения отрицательных чисел). Унарный «плюс» обычно не используется, поскольку числа без знака по умолчанию интерпретируются как положительные, но в принципе его можно писать.

Операторы сравнения

> Когда у общества нет цветовой дифференциации штанов, то нет цели! А когда нет цели — нет будущего!
>
> *из к/ф «Кин-дза-дза»*

Операторы сравнения все бинарные и используются для сравнения значений переменных или литералов. Результатом выражения на основе оператора сравнения является логическое значение (значение типа `bool`). Если соотношение истинно, то результатом является значение `true`, а если соотношение ложно, то результатом является значение `false`. В табл. 2.3 перечислены и кратко описаны операторы сравнения, используемые в языке C#.

Табл. 2.3. Операторы сравнения

Оператор	Описание
<	Оператор *«меньше»*. Значение выражения вида A<B равно true, если значение операнда A меньше значения операнда B. В противном случае значение выражения A<B равно false
<=	Оператор *«меньше или равно»*. Значение выражения вида A<=B равно true, если значение операнда A меньше или равно значению операнда B. В противном случае значение выражения A<=B равно false
>	Оператор *«больше»*. Значение выражения вида A>B равно true, если значение операнда A больше значения операнда B. В противном случае значение выражения A>B равно false
>=	Оператор *«больше или равно»*. Значение выражения вида A>=B равно true, если значение операнда A больше или равно значению операнда B. В противном случае значение выражения A>=B равно false
=	Оператор *«равно»*. Значение выражения вида A==B равно true, если значение операнда A равно значению операнда B. В противном случае значение выражения A==B равно false
!=	Оператор *«не равно»*. Значение выражения вида A!=B равно true, если значение операнда A не равно значению операнда B. В противном случае значение выражения A!=B равно false

Обычно выражения на основе операторов сравнения используются в управляющих инструкциях или в качестве операндов в выражениях на основе логических операторов (если нужно записать сложное условие).

НА ЗАМЕТКУ

Если в выражении на основе оператора сравнения использован символьный операнд, то он преобразуется к числовому виду. Особенности использования операторов сравнения с текстовыми операндами обсуждаются в главе, посвященной работе с текстом.

Логические операторы

— Встань-ка вон там, у той липы.
— Ваше сиятельство, это не липа, это дуб.
— Да? А выглядит как липа...

из к/ф «О бедном гусаре замолвите слово»

Операндами в выражениях на основе *логических операторов* являются значения логического типа (true или false), и результатом таких

выражений также являются значения логического типа. Логические операторы полезны для того, чтобы на основе некоторых простых условий (логических значений) получить новое более сложное условие (логическое). Логические операторы перечислены и кратко описаны в табл. 2.4.

Табл. 2.4. Логические операторы

Оператор	Описание
`&`	Оператор *«логическое и»*. Результатом выражения `A&B` является значение `true`, если оба операнда `A` и `B` равны `true`. Если хотя бы один из операндов равен `false`, то результатом выражения `A&B` является значение `false`
`&&`	Оператор *«логическое и»* (упрощенная форма). Значение выражения вида `A& &B` равно `true`, если оба операнда `A` и `B` равны `true`. В противном случае результат выражения равен `false`. Если при вычислении первого операнда `A` оказалось, что он равен `false`, то значение второго операнда `B` не вычисляется
`|`	Оператор *«логическое или»*. Результатом выражения `A|B` является значение `true`, если хотя бы один из операндов `A` или `B` равен `true`. Если оба операнда равны `false`, то результатом выражения `A|B` является значение `false`
`||`	Оператор *«логическое или»* (упрощенная форма). Значение выражения вида `A||B` равно `true`, если хотя бы один из операндов `A` или `B` равен `true`. В противном случае результат выражения равен `false`. Если при вычислении первого операнда `A` оказалось, что он равен `true`, то значение второго операнда `B` не вычисляется
`^`	Оператор *«логическое исключающее или»*. Результатом выражения `A^B` является значение `true`, если один из операндов `A` или `B` равен `true`, а другой равен `false`. Если оба операнда равны `false` или оба операнда равны `true`, то результатом выражения `A^B` является значение `false`
`!`	Оператор *«логическое отрицание»*. Значение выражения `!A` равно `true`, если значение операнда `A` равно `false`. Если значение операнда `A` равно `true`, то значение выражения `!A` равно `false`

Логические операторы позволяют выполнять четыре операции: *«логическое и»*, *«логическое или»*, *«логические исключающее или»* и *«логическое отрицание»*. Более сложные логические операции выполняются путем комбинированного использования логических операторов. Для выполнения операций *«логическое и»* и *«логическое или»* можно использовать две формы операторов (основную и упрощенную).

Правила вычисления результата для операции *«логическое и»* проиллюстрированы в табл. 2.5.

Табл. 2.5. Операция «*логическое и*» (A&B или A&&B)

A / B	true	false
true	true	false
false	false	false

Если операнды A и B отождествлять с некоторыми условиями, то «сложное» условие A&B (или A&&B) состоит в том, что истинно и условие A, и условие B. Другими словами, значение выражения A&B (или A&&B) будет равно true, только если равно true значение операнда A и одновременно равно true значение операнда B. Очевидно, что если при проверке значения операнда A окажется, что его значение равно false, то результатом логической операции будет false вне зависимости от того, каково значение операнда B. Этот факт принимается во внимание в работе упрощенной формы оператора &&: если при вычислении значения выражения вида A&&B оказалось, что значение операнда A равно false, то значение операнда B вообще не будет вычисляться. При вычислении значения выражения вида A&B значение операнда B вычисляется вне зависимости от того, какое значение операнда A.

НА ЗАМЕТКУ

На первый взгляд может показаться, что различие между операторами & и && «косметическое». Но это не так. Проиллюстрируем это на небольшом примере. Допустим, имеются две целочисленные переменные x и y. Рассмотрим выражение (x!=0)&(y/x>1). Проанализируем, как будет вычисляться значение такого выражения. Сначала вычисляется значение выражения x!=0. Оно равно true, если значение переменной x отлично от нуля. Затем вычисляется значение выражения y/x>1. Оно равно true, если частное от деления y на x больше 1. Проблема в том, что если значение переменной x равно 0, то при вычислении выражения y/x возникнет ошибка (деление на ноль). А вот если вместо выражения (x!=0)&(y/x>1) взять выражение (x!=0)&&(y/x>1), то проблема с делением на ноль снимается автоматически: если значение переменной x равно 0, то значение выражения x!=0 равно false и условие y/x>1 вообще вычисляться не будет.

Как вычисляется результат операции *«логические или»*, демонстрирует табл. 2.6.

Табл. 2.6. Операция *«логическое или»* (`A|B` или `A||B`)

A / B	true	false
true	true	true
false	true	false

«Сложное» условие `A|B` (или `A||B`) состоит в том, что истинно хотя бы одно условие, `A` или `B`. Значением выражений `A|B` или `A||B` является `false` только в том случае, если оба операнда `A` и `B` равны `false`. Поэтому если при вычислении значения операнда `A` окажется, что его значение равно `true`, то в принципе значение второго операнда `B` можно не вычислять — результат от этого не зависит. Именно по такой схеме «работает» упрощенная форма оператора `||`. При использовании оператора `|` значение второго операнда вычисляется вне зависимости от значения первого операнда.

Для операции *«логическое исключающее или»* можно дать такую интерпретацию: результат является истинным, если у операндов разные значения. Более детально это показано в табл. 2.7.

Табл. 2.7. Операция *«логическое исключающее или»* (`A^B`)

A / B	true	false
true	false	true
false	true	false

При одинаковых значениях операндов `A` и `B` значением выражения `A^B` является `false`, а если значения операндов `A` и `B` разные, то результатом выражения является значение `true`.

НА ЗАМЕТКУ

Операцию *«логическое исключающее или»* можно выполнить с помощью операторов `&`, `|` и `!`. Желающие могут проверить, что для логических значений `A` и `B` результаты выражений `A^B` и `(A|B)&(!(A&B))` совпадают.

Оператор `!` для выполнения операции *«логическое отрицание»* — единственный логический унарный оператор. Правила выполнения этой операции иллюстрируются в табл. 2.8.

Табл. 2.8. Операция *«логическое отрицание»* (`!A`)

A	true	false
!A	false	true

Здесь все просто. Если операнд `A` истинный (значение `true`), то результат выражения `!A` ложный (значение `false`). Если операнд `A` ложный (значение `false`), то результат выражения `!A` истинный (значение `true`). И в том и в другом случае значение операнда `A` остается неизменным — то есть выполнение команды вида `!A` не меняет значение операнда `A`.

НА ЗАМЕТКУ

Мы еще будем обсуждать логические операторы в главе, посвященной перегрузке операторов.

Побитовые операторы и двоичные коды

> Варварская игра, дикая местность — меня тянет на родину.
>
> *из к/ф «Формула любви»*

Побитовые операторы предназначены для выполнения операций с целочисленными значениями на уровне их побитового (двоичного) представления. Перед тем как приступить к обсуждению собственно побитовых операторов, кратко рассмотрим способы кодирования чисел с помощью бинарных представлений.

Итак, с некоторой натяжкой можно полагать, что в памяти компьютера числа представлены в виде двоичного кода, в котором есть нули и единицы. Идея записи числа с помощью нулей и единиц принципиально мало чем отличается от способа записи числа посредством цифр от 0 до 9. Допустим, параметры a_0, a_1... a_n — это некоторый набор цифр (то есть каждый такой параметр — это какая-то из цифр 0, 1 и так далее до 9). Мы можем записать с их помощью число $\overline{a_n a_{n-1} \ldots a_2 a_1 a_0}$. Данная запись означает буквально следующее: $\overline{a_n a_{n-1} \ldots a_2 a_1 a_0} = a_0 \times 10^0 + a_1 \times 10^1 + a_2 \times 10^2 \ldots + a_{n-1} \times 10^{n-1} + a_n \times 10^n$. Например, число $123 = 3 \times 10^0 + 2 \times 10^1 + 1 \times 10^2$. А теперь воспользуемся теми же принципами, но вместо десяти цифр (от 0 до 9) будем использовать всего две: 0 и 1. То есть теперь параметры a_0, a_1... a_n могут принимать одно из двух значений (0 или 1). Если число записано в виде последовательности $\overline{a_n a_{n-1} \ldots a_2 a_1 a_0}$, то это будет означать $\overline{a_n a_{n-1} \ldots a_2 a_1 a_0} = a_0 \times 2^0 + a_1 \times 2^1 + a_2 \times 2^2 + \ldots + a_{n-1} \times 2^{n-1} + a_n \times 2^n$. Например, если имеется двоичный код 1011011, то он соответствует десятичному числу $1011011 = 1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 1 \times 2^4 + 0 \times 2^5 + 1 \times 2^6 = 1 + 2 + 8 + 16 + 64 = 91$. Чтобы выполнить обратное преобразование (десятичное число записать двоичным кодом), достаточно число записать в виде суммы слагаемых, являющихся степенями двойки. Например, определим двоичный код для числа 29. Имеем следующее: $29 = 16 + 8 + 4 + 1 = 2^4 + 2^3 + 2^2 + 2^0 = 1 \times 2^0 + 1 \times 2^2 + 1 \times 2^3 + 1 \times 2^4 = 11101$.

НА ЗАМЕТКУ

В приведенных формулах код из нулей и единиц — это двоичное представление числа. Все прочие числа записаны в десятичной системе.

Операции, которые выполняются с десятичными числами, можно выполнять и в двоичном представлении. Например, можно складывать числа в столбик. Для этого достаточно учесть простые правила сложения двоичных чисел: $0 + 0 = 0$, $1 + 0 = 1$, $0 + 1 = 1$, $1 + 1 = 10$. Например, десятичное число 13 имеет двоичный код 1101, а число 9 имеет двоичный код 1001. Складываем эти двоичные коды:

$$\begin{array}{r} 1101 \\ +\ 1001 \\ \hline 10110 \end{array}$$

Двоичный код 10110 соответствует десятичному числу 22. Так и должно быть, поскольку $13 + 9 = 22$.

Все описанное выше касалось положительных чисел. Есть еще один важный вопрос. Связан он с тем, как в памяти компьютера кодируются отрицательные числа. Проблема в том, что для записи отрицательного числа на бумаге достаточно перед таким числом поставить знак «минус». Но компьютер не умеет «писать минус». Он понимает только двоичные коды. Чтобы понять принципы кодирования отрицательных чисел, проведем небольшой мысленный эксперимент.

Допустим, имеется некоторое положительное число A, для которого известен бинарный код. Попробуем выяснить, какой бинарный код должен быть у противоположного по знаку числа $-A$. Мы будем исходить из того, что код числа $-A$ должен быть таким, чтобы выполнялось условие $A + (-A) = 0$.

Выполним побитовую инверсию: в бинарном представлении числа A все нули заменим на единицы, а единицы заменим на нули. То, что получилось, обозначим как $\sim A$. Сложим значения A и $\sim A$. Операцию будем выполнять на уровне бинарного кода. Учтем также, что в компьютере для записи числа выделяется определенное количество битов. Обозначим это количество битов как n. Несложно догадаться, что при вычислении суммы $A + (\sim A)$ мы будем складывать два битовых кода, каждый длиной в n битов. И при складывании битов, находящихся на одинаковых позициях, один бит будет единичный, а другой нулевой. Поэтому в результате вычисления суммы $A + (\sim A)$ получим бинарный код, который состоит из n битов, и каждый бит равен единице. К тому, что получилось, прибавим единицу — то есть вычислим сумму $A + (\sim A) + 1$. Но если к коду из n единиц добавить единицу, то получим бинарный код, в котором самый левый бит единичный, а после него следует n нулей. И вот здесь нужно вспомнить, что речь идет о компьютере, который запоминает только n битов. Поэтому старший единичный бит теряется (для него в памяти просто нет места), и остается код из n нулей. А это код числа 0. Таким образом, компьютер вычисляет указанную сумму как $A + (\sim A) + 1 = 0$. Следовательно, код отрицательного числа $-A$ должен совпадать с кодом выражения $\sim A + 1$. Отсюда мы получаем правило записи двоичного кода отрицательного числа.

- Берем код соответствующего положительного числа и выполняем побитовую инверсию: меняем единицы на нули и нули на единицы.
- К полученному бинарному коду прибавляем единицу. Это и есть код отрицательного числа.

Аналогичным образом выполняется и обратное преобразование, когда по коду отрицательного числа необходимо определить это число в десятичной системе счисления.

- В бинарном коде отрицательного числа выполняем побитовую инверсию.
- К полученному коду прибавляем единицу.
- Переводим полученный код в десятичную систему.
- Дописываем перед полученным числом знак «минус». Это и есть искомое число.

Все эти операции для нас имеют скорее теоретический интерес. Но есть важное практическое последствие. Если взять код положительного числа, то старший бит в таком числе будет нулевым. А вот в коде отрицательного числа старший бит единичный. Таким образом, самый старший бит в бинарном коде числа определяет, положительное это число или отрицательное: у положительных чисел старший бит нулевой, у отрицательных чисел — единичный. Поэтому старший бит обычно называют знаковым битом (он определяет знак числа).

НА ЗАМЕТКУ

В прикладном плане, кроме двоичной, достаточно популярными являются восьмеричная и шестнадцатеричная системы счисления. В восьмеричной системе число записывается с помощью цифр от 0 до 7 включительно. Если в восьмеричной системе число записано в виде последовательности $\overline{a_n a_{n-1}... a_2 a_1 a_0}$ (параметры a_0, a_1... a_n — это цифры от 0 до 7), то в десятичной системе счисления это число определяется так: $\overline{a_n a_{n-1}... a_2 a_1 a_0} = a_0 \times 8^0 + a_1 \times 8^1 + a_2 \times 8^2 + ... + a_{n-1} \times 8^{n-1} + a_n \times 8^n$.
В шестнадцатеричной системе счисления число кодируется с помощью шестнадцати символов: это цифры от 0 до 9 и буквы от A до F, которые обозначают десятичные числа от 10 до 15. Для шестнадцатеричной системы справедлива такая формула перевода в десятичную систему: $\overline{a_n a_{n-1}... a_2 a_1 a_0} = a_0 \times 16^0 + a_1 \times 16^1 + a_2 \times 16^2 + ... + a_{n-1} \times 16^{n-1} + a_n \times 16^n$. В данном случае параметры a_0, a_1... a_n — это цифры от 0 до 9 и буквы от A до F (вместо букв при вычислении суммы подставляются числа от 10 до 15).

После того как мы кратко познакомились со способами бинарного кодирования чисел, можем приступить к обсуждению побитовых операторов. Они перечислены в табл. 2.9.

Табл. 2.9. Побитовые операторы

Оператор	Описание
`&`	Оператор *«побитовое и»*. Результатом выражения `A&B` является число, которое получается сравнением соответствующих битов в представлении чисел `A` и `B`. На выходе получается единичный бит, если оба сравниваемых бита единичные. Если хотя бы один из сравниваемых битов нулевой, в результате получаем нулевой бит
`|`	Оператор *«побитовое или»*. Значение выражения вида `A|B` есть число, получаемое сравнением соответствующих битов в представлении чисел `A` и `B`. На выходе получается единичный бит, если хотя бы один из сравниваемых битов единичный. Если оба сравниваемых бита нулевые, в результате получаем нулевой бит
`^`	Оператор *«побитовое исключающее или»*. Значением выражения `A^B` является число — результат сравнения соответствующих битов в представлении чисел `A` и `B`. На выходе получается единичный бит, если сравниваемые биты различны (один единичный, а другой нулевой). Если биты одинаковые (оба единичные или оба нулевые), то в результате получаем нулевой бит
`~`	Оператор *«побитовая инверсия»*. Значение выражения вида `~A` есть число, битовое представление которого получается заменой в битовом представлении числа `A` нулей на единицы и единиц на нули. Значение операнда `A` при этом не меняется
`<<`	Оператор *«сдвиг влево»*. Результатом выражения `A<<n` является число, которое получается путем сдвига бинарного представления числа `A` на `n` позиций влево. Младшие биты при этом заполняются нулями, а старшие теряются. Значение операнда `A` остается неизменным
`>>`	Оператор *«сдвиг вправо»*. Значение выражения `A>>n` равно числу, которое получается путем сдвига бинарного представления числа `A` на `n` позиций вправо. Младшие биты при этом теряются, а старшие заполняются значением старшего (знакового) бита. Значение операнда `A` остается неизменным

Вообще побитовые операторы напоминают логические, с поправкой на то, что операции выполняются с битами, истинному значению соответствует `1`, а ложному значению соответствует `0`. Например, при вычислении выражения `A&B` на основе оператора *«побитового и»* в результате получается целое число. Его бинарный код вычисляется так: сравниваются биты, находящиеся на одинаковых позициях в представлении чисел `A` и `B`. Правила сравнения даются в табл. 2.10.

Табл. 2.10. Операция *«побитовое и»* (`a&b`) для битов `a` и `b`

a / b	1	0
1	1	0
0	0	0

Если оба бита единичные, то в соответствующем месте в коде для числа-результата записывается единица. Если хотя бы один из двух сравниваемых битов нулевой, то в число-результат записывается нулевой бит.

НА ЗАМЕТКУ

Оператор *«логические и»* и *«побитовое и»* записываются одним и тем же символом `&`. О каком операторе идет речь, можно понять по операндам. Если операнды числовые, то это побитовый оператор. Если операнды логические, то это логический оператор. Это же замечание относится и к рассматриваемым далее операторам `|` и `^`.

Как пример простой операции с использованием оператора *«побитового и»* `&` рассмотрим следующий блок программного кода:

```
byte A=13, B=25, C;
C=(byte)(A&B);
```

В данном случае имеем дело с тремя переменными `A`, `B` и `C` типа `byte`. Это значит, что значение каждой из переменных запоминается с помощью 8 битов.

НА ЗАМЕТКУ

Хотя переменные `A` и `B` объявлены как относящиеся к типу `byte`, при выполнении операции `A&B` из-за автоматического приведения типа результат вычисляется как `int`-значение. Это значение мы пытаемся записать в переменную `C` типа `byte`. Поэтому здесь необходима инструкция `(byte)` явного приведения типа. Это замечание относится и к нескольким следующим мини-примерам.

Десятичное число `13` в двоичном 8-битовом коде записывается как `00001101`, а числу `25` соответствует бинарный код `00011001`.

ПОДРОБНОСТИ

Определить бинарные коды для чисел 13 и 25 можно, если учесть что $13 = 8 + 4 + 1 = 2^3 + 2^2 + 2^0 = 0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$, а также $25 = 16 + 8 + 1 = 2^4 + 2^3 + 2^0 = 0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$.

Операция *«побитовое и»* выполняется так:

$$\begin{array}{r} 00001101 \\ \& \ 00011001 \\ \hline 00001001 \end{array}$$

Следовательно, в результате получаем бинарный код `00001001`. Несложно проверить, что это код числа $2^3 + 2^0 = 9$. Таким образом, при выполнении указанного выше программного кода значения переменных будут следующими: значение переменной `A` равно `13`, значение переменной `B` равно `25`, а значение переменной `C` равно `9`.

Если используется оператор *«побитовое или»* `|`, то при сравнении двух битов в результате получаем единичный бит, если хотя бы один из сравниваемых битов единичный. Правила выполнения этой операции проиллюстрированы в табл. 2.11.

Табл. 2.11. Операция *«побитовое или»* (`a | b`) для битов `a` и `b`

a / b	1	0
1	1	1
0	1	0

Рассмотрим, как операция *«побитовое или»* выполняется с уже знакомыми для нас переменными:

```
byte A=13, B=25, C;
C=(byte)(A|B);
```

В соответствии с правилами выполнения операции получаем следующий результат:

$$\begin{array}{r} 00001101 \\ | \ 00011001 \\ \hline 00011101 \end{array}$$

Бинарный код `00011101` соответствует десятичному числу `29`, поскольку имеет место соотношение $2^4 + 2^3 + 2^2 + 2^0 = 29$. Таким образом, переменная `C` в данном случае будет иметь значение `29`.

Если выполняется операция *«побитовое исключающее или»*, то двоичные коды для двух исходных чисел сравниваются с использованием такой

схемы: если на одинаковых позициях в битовых представлениях чисел значения разные (то есть один бит единичный, а другой нулевой), то на «выходе» получаем единицу. Если у сравниваемых чисел на одинаковых позициях находятся одинаковые биты (то есть биты с одинаковыми значениями), в числе-результате на соответствующем месте будет нулевой бит (бит с нулевым значением). Правила выполнения операции *«побитовое исключающее или»* проиллюстрированы в табл. 2.12 для двух отдельных битов.

Табл. 2.12. Операция *«побитовое исключающее или»* (`a^b`) для битов `a` и `b`

a / b	1	0
1	0	1
0	1	0

Результат применения операции *«побитовое исключающее или»* к числовым значениям `13` и `25` приводит к следующему результату:

$$\begin{array}{r} 00001101 \\ \wedge\ 00011001 \\ \hline 00010100 \end{array}$$

Мы получаем бинарный код `00010100`, который соответствует десятичному числу `20` (поскольку $2^4 + 2^2 = 20$). Допустим, мы имеем дело, например, с таким блоком программного кода:

```
byte A=13, B=25, C;
C=(byte)(A^B);
```

В результате его выполнения переменная `C` получит значение `20`.

Оператор *«побитовая инверсия»* ~ является унарным. Побитовая инверсия выполняется просто: в битовом представлении числа нули меняются на единицы, а единицы меняются на нули. Это правило, по которому вычисляется результат. Операнд при этом не меняется. В табл. 2.13 показано, как операция побитовой инверсии выполняется для одного отдельного бита.

Табл. 2.13. Операция *«побитовая инверсия»* (`~a`) для бита `a`

a	1	0
~a	0	1

Например, имеется такой блок программного кода:

```
byte A, B;
A=13;
B=(byte)~A;
```

Вопрос состоит в том, какие значения будут у целочисленных переменных A и B после выполнения указанных команд? Ответ, в принципе, простой: значение переменной A равно 13, а значение переменной B равно 242. Самая простая ситуация с переменной A. Ей присваивается значение 13. Это число в десятичной системе. В двоичной системе код значения переменной A имеет вид 00001101. Но в данном случае это не столь важно, поскольку при выполнении команды B=(byte)~A значение переменной A не меняется: побитовая инверсия применяется для вычисления результата выражения ~A, но сам операнд A не меняется. Поэтому переменная A остается со своим, ранее присвоенным значением (число 13).

ПОДРОБНОСТИ

Напомним, что значение типа byte записывается с использованием 8 битов и записанное таким образом число интерпретируется как неотрицательное число. Поэтому старший бит в представлении byte-числа знаковым не является.

Если применить операцию побитового инвертирования к коду 00001101 числа 13, то в бинарном 8-битовом представлении получим код 11110010. Этот код соответствует числу $2^7 + 2^6 + 2^5 + 2^4 + 2^1 = 128 + 64 + 32 + 16 + 2 = 242$. Таким образом, переменная B получает значение 242. Но еще раз подчеркнем, что в данном случае важно, что тип byte предназначен для реализации неотрицательных чисел. Если мы вместо типа byte воспользуемся типом sbyte, результат побитовой инверсии будет иным. Точнее, он будет по-другому интерпретироваться. Допустим, имеется такой код:

```
sbyte A, B;
A=13;
B=(sbyte)~A;
```

Вопрос состоит в том, каково значение переменной B? Строго говоря, как и в предыдущем случае, значение переменной B будет представлено

кодом 11110010. Но теперь, поскольку это sbyte-значение, код интерпретируется как число со знаком. Так как старший бит единичный, то число отрицательное. Чтобы понять, что это за число, мы должны:

- выполнить побитовую инверсию;
- прибавить к результату единицу;
- перевести код в десятичное число;
- дописать знак «минус» перед числом.

Поскольку нам предстоит инвертировать то, что было получено инвертированием, мы сначала получим бинарный код для числа 13, затем прибавим к этому числу 1 (получим 14) и, дописав знак «минус», получаем значение –14 для переменной B.

Еще две побитовые операции, которые нам осталось обсудить — сдвиг влево и сдвиг вправо. Например, рассмотрим следующий программный код:

```
byte A=13, B, C;
B=(byte)(A<<2)
C=(byte)(A>>2);
```

Чтобы получить значение переменной B, мы должны взять код 00001101 для числа 13 (значение переменной A) и выполнить сдвиг этого кода влево на две позиции. В результате получаем код 00110100. Это код числа 52 — значение переменной B. Значение переменной A остается неизменным.

НА ЗАМЕТКУ

Сдвиг влево на одну позицию бинарного кода числа эквивалентен умножению числа на 2. Сдвиг бинарного кода положительного числа на одну позицию вправо эквивалентен целочисленному делению этого числа на 2.

Для вычисления значения переменной C бинарный код значения переменной A сдвигается вправо на две позиции. В результате из кода 00001101 получаем код 00000011, что соответствует числу 3. Это и есть значение переменной C.

Оператор присваивания

— Нет! На это я пойтить не могу!
— Товарищ старший лейтенант...
— Нет! Мне надо носоветоваться с шефом! С начальством!

из к/ф «Бриллиантовая рука»

С оператором присваивания = мы уже знакомы: он используется для присваивания значений переменным. Обычно используются команды вида `переменная=значение`. Напомним, что в результате выполнения такой команды `переменная`, указанная слева от оператора присваивания =, получает `значение`, указанное справа от оператора присваивания. Но особенность оператора присваивания в том, что он возвращает значение. Поэтому у выражения вида `переменная=значение` есть значение, и такое выражение можно использовать в более сложном выражении. Проще говоря, в одной команде может быть больше одного оператора присваивания.

Самая простая ситуация — когда сразу нескольким переменным присваивается значение. Скажем, следующий блок программного кода имеет смысл и является корректным:

```
int A, B, C;
A=B=C=123;
```

Здесь три целочисленные переменные `A`, `B` и `C` значением получают число `123`. Объяснение простое. Команда `A=B=C=123` представляет собой инструкцию, которой переменной `A` значением присваивается выражение `B=C=123`. Выражение `B=C=123` — это команда, которой переменной `B` присваивается значением выражение `C=123`. Командой `C=123` переменной `C` присваивается значение `123`. И само это выражение `C=123` имеет значение — это значение, присваиваемое переменной `C` (то есть `123`). Следовательно, переменная `B` получает значение `123`, и значением выражения `B=C=123` также является число `123`. Это значение присваивается переменной `A`. В итоге все три переменные (`A`, `B` и `C`) получают значение `123`.

Команда с несколькими операторами присваивания может быть более замысловатой. Как пример рассмотрим следующий блок программного кода:

```
int A, B, C;
A=(A=(A=3)+(B=5)-(C=7-B))+(A=A+(B=B+2)*(C=C-3));
```

После выполнения этого кода у переменной A будет значение 5, у переменной B будет значение 7, а переменная C получит значение –1. Чтобы понять, почему так происходит, проанализируем команду `A=(A=(A=3)+(B=5)-(C=7-B))+(A=A+(B=B+2)*(C=C-3))`. В этой команде значением переменной A присваивается выражение `(A=(A=3)+(B=5)-(C=7-B))+(A=A+(B=B+2)*(C=C-3))`. Данное выражение представляет собой сумму двух операндов: `(A=(A=3)+(B=5)-(C=7-B))` и `(A=A+(B=B+2)*(C=C-3))`. Операнды вычисляются слева направо. Первым вычисляется операнд `(A=(A=3)+(B=5)-(C=7-B))`. После него вычисляется операнд `(A=A+(B=B+2)*(C=C-3))`.

Операнд `(A=(A=3)+(B=5)-(C=7-B))` представляет собой выражение, в котором переменной A значением присваивается выражение `(A=3)+(B=5)-(C=7-B)`. В нем три операнда: `A=3`, `B=5` и `C=7-B`. Они вычисляются последовательно слева направо. В результате сначала переменная A получает значение 3 (и это же значение выражения `A=3`), переменная B получает значение 5 (значение выражения `B=5`), а переменная C получает значение 2 (оно же — значение выражения `C=7-B`). Тогда результат выражения `(A=3)+(B=5)-(C=7-B)` равен 6 (3 плюс 5 минус 2). А поскольку это выражение присваивается значением переменной A (операнд `(A=(A=3)+(B=5)-(C=7-B))`), то значение переменной A становится равным 6.

При вычислении значения второго операнда `(A=A+(B=B+2)*(C=C-3))` значением переменной A присваивается сумма двух выражений: A и `(B=B+2)*(C=C-3)`. Как мы выяснили выше, текущее значение переменной A равно 6. При вычислении выражения `B=B+2` (учитывая, что текущее значение переменной B равно 5) получается, что значение переменной B равно 7 и это же будет значение выражения `B=B+2`. Для выражения `C=C-3` получаем значение –1 (оно же значение переменной C): в данном случае нужно учесть, что на предыдущем этапе переменная C получила значение 2. В итоге значение выражения `(B=B+2)*(C=C-3)` равно –7. Тогда второй операнд `(A=A+(B=B+2)*(C=C-3))` равен –1 (6 плюс –7). Следовательно, значение выражения `(A=(A=3)+(B=5)-(C=7-B))+(A=A+(B=B+2)*(C=C-3))` равно 5 (6 плюс –1). Это новое значение переменной A. Таким образом, переменная A получила значение 5, переменная B получила значение 7, а переменная C получила значение –1.

НА ЗАМЕТКУ

Сложные команды и выражения лучше разбивать на несколько простых. Такой программный код легче читать, и меньше вероятность сделать ошибку.

Сокращенные формы операции присваивания

Отлично, отлично! Скромненько и со вкусом!

из к/ф «Бриллиантовая рука»

Допустим, что ☺ — это некоторый бинарный оператор (арифметический или побитовый). Если нам необходимо выполнить операцию вида `A=A ☺ B` (то есть вычислить значение выражения `A ☺ B` на основе оператора ☺, а затем результат выражения присвоить значением переменной `A`), то такая операция может быть реализована с помощью команды `A☺=B`. Например, вместо выражения `A=A+B` можно использовать команду `A+=B`. Аналогично команду `A=A<<B` можно записать как `A<<=B` или вместо выражения `A=A^B` использовать выражение `A^=B`. Во всех перечисленных случаях говорят о сокращенной форме операции присваивания.

Помимо очевидного «эстетического» удобства, сокращенные формы операции присваивания имеют еще одно несомненное преимущество. Чтобы понять, в чем оно состоит, рассмотрим небольшой блок программного кода, в котором переменная `A` в итоге получает значение `15`:

```
byte A=10, B=5;
A=(byte)(A+B);
```

В нем проиллюстрирована уже знакомая нам ситуация: чтобы присвоить переменной `A` новое значение, необходимо использовать автоматическое приведение типа.

НА ЗАМЕТКУ

Переменные `A` и `B` объявлены как относящиеся к типу `byte`. Но при вычислении выражения `A+B` выполняется автоматическое расширение до типа `int`. Поэтому, если мы хотим выражение `A+B` присвоить значением переменной `A` (типа `byte`), то перед выражением `A+B`

следует указать инструкцию (byte) — то есть выполнить приведение int-значения к byte-значению.

Но если мы используем сокращенную форму операции присваивания, то явное приведение типа выполнять не нужно:

```
byte A=10, B=5;
A+=B;
```

В данном случае переменная A также получит значение 15. Причем инструкцию (byte) мы не использовали — в этом нет необходимости. Объяснение состоит в том, что команда вида A☺=B в действительности вычисляется немного сложнее, чем выполнение команды A=A ☺ B. Точнее, если результат выражения A ☺ B может быть автоматически преобразован к типу переменной A, то проблемы нет вообще. Но если для результата выражения A ☺ B автоматического преобразования к типу переменной A нет, то схема иная. А именно если результат выражения A ☺ B может быть явно преобразован к типу переменной A, а тип переменной B при этом может быть неявно преобразован к типу переменной A, то преобразование результата выражения A ☺ B к типу переменной A выполняется автоматически (без явного указания инструкции приведения типа). Проще говоря, если результат выражения A ☺ B можно преобразовать (с помощью процедуры явного приведения типа) к типу переменной A (*первое условие*) и если тип переменной B такой, что допускается неявное преобразование к типу переменной A (*второе условие*), то выражение A☺=B вычисляется как A=(тип A)(A ☺ B). После вычисления выражения A ☺ B полученное значение автоматически преобразуется к типу переменной A, а затем выполняется операция присваивания. Например, проанализируем команду A+=B, в которой переменные A и B относятся к типу byte. Значение выражения A+B — это число типа int. А присваивать значение нужно переменной типа byte. Тип int в тип byte автоматически не преобразуется. Но такое преобразование можно выполнить явно. То есть первое условие выполнено. Далее, переменная B относится к типу byte. Необходимо, чтобы для типа переменной B существовала возможность неявного преобразования к типу переменной A. Но в данном случае эти типы просто совпадают, поэтому второе условие тоже выполнено. Следовательно, результат выражения A+B автоматически преобразуется к типу переменной A (тип byte).

Тернарный оператор

— Лелик, это же не эстетично...
— Зато дешево, надежно и практично!

из к/ф «Бриллиантовая рука»

В языке C# есть один оператор, который используется с тремя операндами. Такой оператор называют *тернарным*. По большому счету, это «миниатюрная» версия *условного оператора*, поскольку значение выражения на основе тернарного оператора зависит от истинности или ложности некоторого *условия*.

> **НА ЗАМЕТКУ**
>
> Здесь и далее под условием будем подразумевать любое логическое выражение — то есть выражение, которое может иметь значение `true` (истинное условие) или `false` (ложное условие).

Синтаксис тернарного оператора такой (жирным шрифтом выделены ключевые элементы шаблона):

```
условие?значение: значение
```

Сначала указывается условие (это первый операнд) — любое логическое выражение. Затем следует вопросительный знак `?` — это синтаксический элемент тернарного оператора. После вопросительного знака указывается некоторое значение (второй операнд). Далее следует двоеточие `:` (синтаксический элемент тернарного оператора) и после двоеточия — еще одно значение (третий операнд). Если условие истинно (значение первого операнда равно `true`), то значением всего выражения является значение второго операнда (значение, указанное после вопросительного знака `?`). Если условие ложно (значение первого операнда равно `false`), то значение всего выражения определяется значением третьего операнда (значение, указанное после двоеточия `:`).

> **НА ЗАМЕТКУ**
>
> Тернарный оператор иногда упоминают как оператор `?:`.

Например, при выполнении команды `num=(x<0)&1:2` переменная `num` получает значение `1`, если значение переменной `x` меньше `0`, и значение `2`, если значение переменной `x` не меньше `0`.

Приоритет операторов

> Дайте мне поручение, а уж особым я его как-нибудь и сам сделаю.
>
> *из к/ф «О бедном гусаре замолвите слово»*

При вычислении сложных выражений, в которых используются различные операторы, порядок выполнения операций определяется *приоритетом операторов*. Подвыражения на основе операторов с более высоким приоритетом вычисляются ранее по сравнению с подвыражениями на основе операторов с более низким приоритетом. Если у операторов одинаковый приоритет, то подвыражения вычисляются в естественном порядке, слева направо. В табл. 2.14 основные операторы языка C# сгруппированы по уровню приоритета (от самых приоритетных до наименее приоритетных).

Табл. 2.14. Приоритет операторов

Приоритет	Операторы
1	Круглые скобки `()`, квадратные скобки `[]`, оператор *«точка»* `.`, операторы инкремента `++` и декремента `--` в постфиксной форме (то есть в выражениях вида `A++` и `A--`)
2	Унарный оператор *«плюс»* `+`, унарный оператор *«минус»* `-`, оператор побитовой инверсии `~`, оператор логического отрицания `!`, операторы инкремента `++` и декремента `--` в префиксной форме (то есть в выражениях вида `++A` и `--A`), операция явного приведения типа
3	Оператор умножения `*`, оператор деления `/`, оператор вычисления остатка от деления `%`
4	Оператор сложения `+`, оператор вычитания `-`
5	Оператор побитового сдвига вправо `>>`, оператор побитового сдвига влево `<<`
6	Оператор сравнения *«больше»* `>`, оператор сравнения *«больше или равно»* `>=`, оператор сравнения *«меньше»* `<`, оператор сравнения *«меньше или равно»* `<=`
7	Оператор сравнения *«равно»* `==`, оператор сравнения «не равно» `!=`
8	Оператор *«побитовое и»* `&` (также *«логическое и»* в полной форме)
9	Оператор *«побитовое исключающее или»* `^` (также *«логическое исключающее или»*)
10	Оператор *«побитовое или»* `|` (также *«логическое или»* в полной форме)

11	Оператор *«логическое и»* в упрощенной форме `&&`
12	Оператор *«логическое или»* в упрощенной форме `||`
13	Тернарный оператор `?:`
14	Оператор присваивания `=`, операторы сокращенного присваивания `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=` и `|=`

Для изменения порядка вычисления выражений можно использовать круглые скобки.

НА ЗАМЕТКУ

Помимо своего прямого назначения — изменения порядка вычисления подвыражений — круглые скобки облегчают чтение программного кода. Так что их нередко используют и в «декоративных» целях, когда прямой необходимости в этом нет.

Примеры программ

Я тебя полюбил — я тебя научу.

из к/ф «Кин-дза-дза»

Далее мы рассмотрим несколько очень простых иллюстраций к тому, как базовые операторы языка C# могут использоваться на практике. Сначала мы создадим программу, предназначенную для проверки введенного пользователем числа на предмет четности/нечетности.

НА ЗАМЕТКУ

Число называется четным, если оно делится на 2 — то есть если при делении на 2 в остатке получаем 0. Если число на 2 не делится, то оно называется нечетным.

Программа выполняется просто: появляется диалоговое окно с полем ввода, в которое пользователю предлагается ввести целое число. Далее появляется еще одно диалоговое окно с сообщением о том, что пользователь ввел четное или нечетное число. Код программы представлен в листинге 2.1.

Листинг 2.1. Проверка числа на четность/нечетность

```
using System;
using System.Windows.Forms;
using Microsoft.VisualBasic;
class OddEvenDemo{
   static void Main(){
      // Целочисленные переменные:
      int number, reminder;
      // Считывание целого числа:
      number=Int32.Parse(
         Interaction.InputBox(
            // Текст в окне:
            "Введите целое число:",
            // Название окна:
            "Проверка")
      );
      // Вычисляется остаток от деления на 2:
      reminder=number%2;
      string txt="Вы ввели ";
      // Использован тернарный оператор:
      txt+=(reminder==0?"четное":"нечетное")+" число!";
      MessageBox.Show(txt);
   }
}
```

В программе используются две целочисленные переменные `number` и `reminder` — соответственно, для запоминания введенного пользователем числа и остатка от деления этого числа на 2. Для считывания числа использован статический метод `InputBox()` класса `Interaction` (для использования этого класса в заголовке программы взята инструкция `using Microsoft.VisualBasic`). Аргументами методу передается два текстовых значения: текст, отображаемый над полем ввода, и название (заголовок) для окна. Метод `InputBox()` значением возвращает введенное пользователем значение, но значение возвращается

в текстовом формате. Для преобразования текстового представления числа в целое число используем статический метод `Parse()` из структуры `Int32` (структура доступна после подключения пространства имен `System` командой `using System`). В частности, инструкция вызова метода `InputBox()` передана аргументом методу `Parse()`. Результат преобразования присваивается значением переменной `number`.

Командой `reminder=number%2` в переменную `reminder` записывается остаток от деления значения переменной `number` на `2`. Переменная `reminder` в принципе может принимать всего два значения: `0` при четном значении переменной `number` и `1` при нечетном значении переменной `number`.

Текстовая переменная `txt` объявляется с начальным значением `"Вы ввели "`. Нам эта переменная нужна для того, чтобы записать в нее текст, который затем будет отображаться во втором диалоговом окне. Следующей командой `txt+=(reminder==0?"четное":"нечетное")+" число!"` к текущему значению этой переменной дописывается недостающая часть. Здесь в подвыражении в правой части мы использовали тернарный оператор. Речь об инструкции `reminder==0?"четное":"нечетное"`. Значение этого выражения определяется так: если значение переменной `reminder` равно `0`, то значение выражения равно `"четное"`. В противном случае (если значение переменной `reminder` отлично от `0`) значение выражения равно `"нечетное"`. Таким образом, в текстовое значение переменной `txt` включается слово `"четное"` или `"нечетное"` в зависимости от значения переменной `reminder`. После того как значение переменной `txt` сформировано, командой `MessageBox.Show(txt)` отображается диалоговое окно с соответствующим сообщением (для использования класса `MessageBox` используем инструкцию `using System.Windows.Forms` в заголовке программы).

При запуске программы на выполнение появляется окно с текстовым полем. На рис. 2.1 показано такое окно с числом `123` в поле ввода.

Рис. 2.1. В поле введено нечетное число

После щелчка по кнопке **ОК** появляется новое окно с сообщением о том, что число нечетное. Окно показано на рис. 2.2.

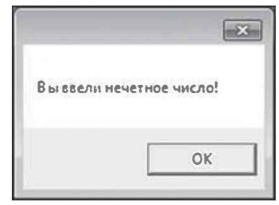


Рис. 2.2. Окно с сообщением о том, что было введено нечетное число

На рис. 2.3 показано начальное диалоговое окно с числом 124 в поле ввода.

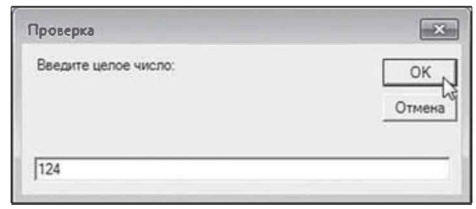


Рис. 2.3. В поле введено четное число

Если так, то вторым появится окно с сообщением о том, что пользователь ввел четное число. Окно показано на рис. 2.4.

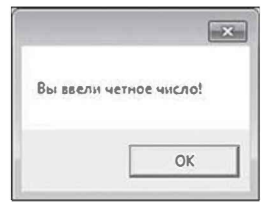


Рис. 2.4. Окно с сообщением о том, что было введено четное число

При этом стоит также заметить, что если пользователь в поле ввода в первом окне введет не число или вместо кнопки **ОК** щелкнет кнопку **Отмена** (или закроет окно с помощью системной пиктограммы), то в программе произойдет ошибка. В принципе, подобные ситуации легко обрабатываются. Но пока что нас этот вопрос занимать не будет.

В следующей программе для числа, введенного пользователем, определяется количество сотен (в десятичном представлении числа это третья цифра справа). В листинге 2.2 представлен код программы.

Листинг 2.2. Количество сотен в числе

```
using System;
using System.Windows.Forms;
using Microsoft.VisualBasic;
class HundredsDemo{
   static void Main(){
      // Целочисленные переменные:
      int number, hundreds;
      // Считывание целого числа:
      number=Int32.Parse(
         Interaction.InputBox(
            // Надпись над полем ввода:
            "Введите целое число:",
            // Заголовок окна:
            "Количество сотен")
      );
      // Количество сотен в числе (для целочисленных
      // операндов деление выполняется нацело):
      hundreds=number/100%10;
      // Текстовая переменная:
      string txt="В этом числе "+hundreds+" сотен!";
      // Отображение окна с сообщением
      // (аргументы метода – сообщение и заголовок окна):
      MessageBox.Show(txt,"Сотни");
   }
}
```

Эта программа напоминает предыдущую — как минимум в части считывания целочисленного значения. Число, введенное пользователем, записывается в переменную `number`. Затем выполняется команда `hundreds=number/100%10`, благодаря чему в переменную `hundreds` записывается количество сотен в числе. Вычисления выполняются так: значение переменной `number` делится на `100`. Здесь нужно учесть, что

для целочисленных операндов выполняется целочисленное деление. Поэтому деление числа на `100` означает фактически отбрасывание двух последних цифр в представлении числа. Как следствие, третья справа цифра в значении переменной `number` «перемещается» на последнюю позицию (первую справа).

> **НА ЗАМЕТКУ**
>
> Значение переменной `number` не меняется. Речь идет о значении выражения `number/100` по сравнению со значением переменной `number`.

Далее вычисляется остаток от деления на `10` значения выражения `number/100` (выражение `number/100%10`). Результат записывается в переменную `hundreds`.

В программе объявляется текстовая переменная `txt` со значением `"В этом числе "+hundreds+" сотен!"`, в котором использована переменная `hundreds`. Переменную `txt` используем в команде `MessageBox.Show(txt,"Сотни")` для отображения окна с сообщением (первый аргумент определяет текст сообщения, а второй задает заголовок окна).

На рис. 2.5 показано окно с полем, содержащим введенное пользователем число.

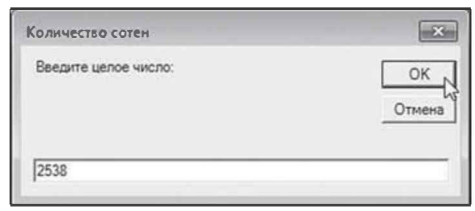


Рис. 2.5. В поле введено целое число для определения количества сотен в числе

После щелчка по кнопке **ОК** появляется окно с сообщением, показанное на рис. 2.6.

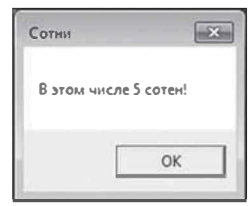


Рис. 2.6. Окно с сообщением о количестве сотен в числе

В данном случае мы ввели число 2538 и программа вычисляет корректное значение 5 для количества сотен в числе.

Резюме

> Только быстро. А то одна секунда здесь — это полгода там!
>
> *из к/ф «Кин-дза-дза»*

- При объявлении переменной указывается ее тип и название. Несколько переменных одного типа может быть объявлено одновременно. При объявлении переменной можно указать ее значение. Константы объявляются с ключевым словом `const`.
- В языке C# существует несколько базовых типов. Целочисленные типы данных `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` и `ulong`. Отличаются эти типы размером выделяемой памяти и возможностью/невозможностью использовать отрицательные числа. Действительные числа реализуются с помощью типов `float` и `double`. Для финансовых расчетов используется тип `decimal`. Символьные значения реализуются через тип `char`. Логические значения реализуются как значения типа `bool`.
- Целочисленные литералы по умолчанию реализуются как `int`-значения, действительные числовые литералы реализуются как `double`-значения, символьные литералы реализуются как `char`-значения, текст реализуется в виде объектов класса `string`.
- Если в выражении используются значения разных (но совместимых) типов, то применяется автоматическое преобразование типов. Можно также выполнять явное приведение типов.
- Для выполнения основных операций применяются специальные операции. В C# есть четыре группы операторов: арифметические (*«сложение»* `+`, *«вычитание»* `-`, *«умножение»* `*`, *«деление»* `/`, *«остаток от деления»* `%`, *«инкремент»* `++`, *«декремент»* `--`), логические (*«логические и»* `&` и `&&`, *«логические или»* `|` и `||`, *«логическое исключающее или»* `^`, *«логическое отрицание»* `!`), операторы сравнения (*«меньше»* `<`, *«меньше или равно»* `<=`, *«больше»* `>`, *«больше или равно»* `>=`, *«равно»* `==`, *«не равно»* `!=`), а также побитовые операторы (*«побитовое и»* `&`, *«побитовое или»* `|`, *«побитовое исключающее или»* `^`,

«побитовая инверсия» ~, *«сдвиг влево»* <<, *«сдвиг вправо»* >>). В C# есть один тернарный оператор ?:, представляющий собой упрощенную форму условного оператора. Для арифметических и побитовых операторов есть сокращенные формы операции присваивания (*=, /=, %=, +=, -=, <<=, >>=, &=, ^= и |=). Оператор присваивания = в языке C# возвращает значение.

Задания для самостоятельной работы

Хочешь поговорить — нлати еще чатл.

из к/ф «Кин-дза-дза»

1. Напишите программу, которая проверяет, делится ли введенное пользователем число на 3.

2. Напишите программу, которая проверяет, удовлетворяет ли введенное пользователем число следующим критериям: при делении на 5 в остатке получается 2, а при делении на 7 в остатке получается 1.

3. Напишите программу, которая проверяет, удовлетворяет ли введенное пользователем число следующим критериям: число делится на 4, и при этом оно не меньше 10.

4. Напишите программу, которая проверяет, попадает ли введенное пользователем число в диапазон от 5 до 10 включительно.

5. Напишите программу, которая проверяет, сколько тысяч во введенном пользователем числе (определяется четвертая цифра справа в десятичном представлении числа).

6. Напишите программу, которая проверяет вторую справа цифру в восьмеричном представлении числа, введенного пользователем. Число вводится в десятичном (обычном) представлении.

7. Напишите программу, которая вычисляет третий бит справа в двоичном представлении числа, введенного пользователем. Число вводится в десятичном (обычном) представлении. В программе используйте оператор побитового сдвига.

8. Напишите программу, в которой для введенного пользователем числа в бинарном представлении третий бит устанавливается равным единице.

9. Напишите программу, в которой для введенного пользователем числа в бинарном представлении четвертый бит устанавливается равным нулю.

10. Напишите программу, в которой для введенного пользователем числа в бинарном представлении значение второго бита меняется на противоположное (исходное нулевое значение бита меняется на единичное, а исходное единичное значение бита меняется на нулевое).

Глава 3
УПРАВЛЯЮЩИЕ ИНСТРУКЦИИ

> Значит, такое предложение: сейчас мы нажимаем на контакты и неремещаемся к вам. Но если эта машинка не сработает, тогда уж вы с нами переместитесь, куда мы вас переместим!
>
> *из к/ф «Кин-дза-дза»*

В этой главе мы рассмотрим вопросы, связанные с использованием управляющих инструкций. Более конкретно, мы сосредоточим внимание на следующих конструкциях языка C#:

- познакомимся с условным оператором `if`;
- научимся использовать оператор выбора `switch`;
- узнаем, как используется оператор цикла `while`;
- выясним, в чем особенности оператора цикла `do-while`;
- убедимся в гибкости и эффективности оператора цикла `for`;
- познакомимся с инструкцией безусловного перехода `goto`;
- составим некоторое представление о системе обработки исключений.

Теперь рассмотрим более детально означенные выше темы.

Условный оператор if

> Или ты сейчас же отдаешь нам эту КЦ, или меньше чем за семь коробков мы тебя на Землю не положим!
>
> *из к/ф «Кин-дза-дза»*

Условный оператор позволяет выполнять разные блоки команд в зависимости от истинности или ложности некоторого *условия* (выражение со значением логического типа). Работает это так: сначала вычисляется значение некоторого логического выражения (условие). Если значение выражения равно `true` (условие истинно), выполняется определенный блок команд. Если значение выражения равно `false` (условие ложно), выполняется другой блок команд.

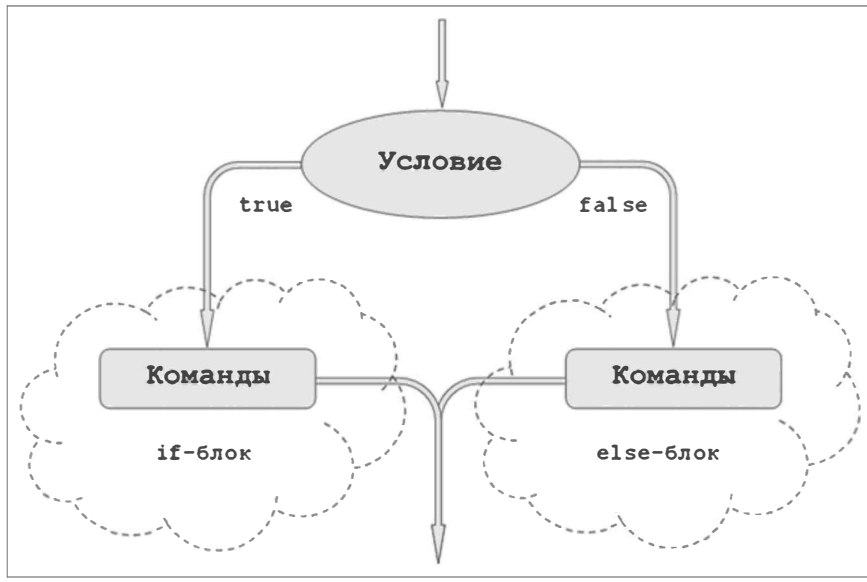


Рис. 3.1. Принципы выполнения условного оператора

Описывается условный оператор достаточно просто: указывается ключевое слово `if`, после которого в круглых скобках следует условие, проверяемое при выполнении условного оператора. Блок команд, выполняемых при истинном условии, указывается в фигурных скобках сразу после ключевого слова `if` с условием (`if`-блок). Команды, предназначенные для выполнения в случае ложного условия, указываются

в фигурных скобках после ключевого слова `else` (`else`-блок). Шаблон описания условного оператора приведен ниже (жирным шрифтом выделены ключевые элементы шаблона):

```
if(условие){
   // Команды – если условие истинно
}
else{
   // Команды – если условие ложно
}
```

Принципы выполнения условного оператора также проиллюстрированы в схеме на рис. 3.1.

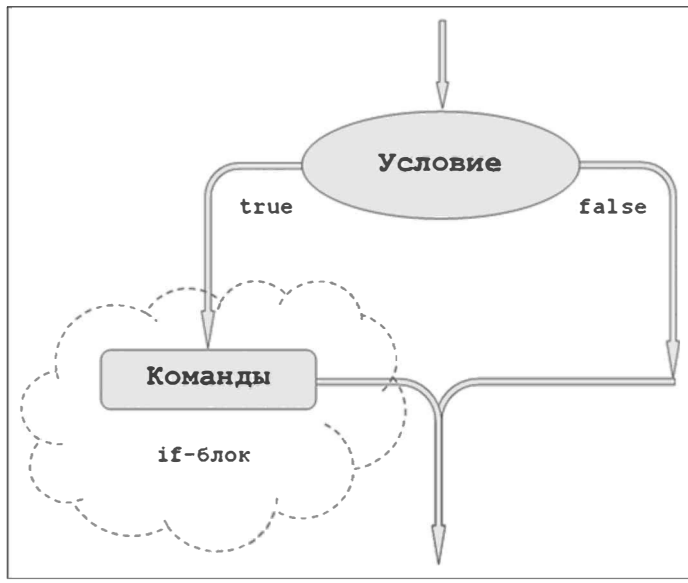


Рис. 3.2. Выполнение условного оператора в упрощенной форме

Если блок состоит всего из одной команды, то фигурные скобки для выделения блока можно не использовать.

У условного оператора есть упрощенная форма, в которой отсутствует `else`-блок. Если так, то оператор выполняется следующим образом: проверяется условие, и, если оно истинно, выполняются команды в `if`-блоке. Если условие ложно, то ничего не происходит — выполняется команда, следующая после условного оператора. Шаблон описания

условного оператора в упрощенной форме такой (жирным шрифтом выделены ключевые элементы шаблона):

```
if(условие){
   // Команды – если условие истинно
}
```

Как выполняется условный оператор в упрощенной форме, иллюстрирует схема на рис. 3.2.

Небольшой пример, иллюстрирующий работу условного оператора, представлен в листинге 3.1.

Листинг 3.1. Использование условного оператора

```
using System.Windows.Forms;
using Microsoft.VisualBasic;
class UsingIfDemo{
   static void Main(){
      // Переменная для определения типа пиктограммы:
      MessageBoxIcon icon;
      // Переменные для определения текста сообщения,
      // заголовка окна и имени пользователя:
      string msg, title, name;
      // Считывание имени пользователя:
      name=Interaction.InputBox(
         // Текст над полем ввода:
         "Как Вас зовут?",
         // Название окна:
         "Знакомимся");
      // Проверка введенного пользователем текста:
      if(name==""){ // Если текст не введен
         // Пиктограмма ошибки:
         icon=MessageBoxIcon.Error;
         // Текст сообщения:
         msg="Очень жаль, что мы не познакомились!";
```

```
            // Заголовок окна:
            title="Знакомство не состоялось";
        }
        else{ // Если текст введен
            // Информационная пиктограмма:
            icon=MessageBoxIcon.Information;
            // Текст сообщения:
            msg="Очень приятно, "+name+"!";
            // Заголовок окна:
            title="Знакомство состоялось";
        }
        // Отображение сообщения (аргументы – текст
        // сообщения, заголовок, кнопки и пиктограмма):
        MessageBox.Show(msg, title, MessageBoxButtons.OK, icon);
    }
}
```

Программа простая: отображается окно с полем ввода, и пользователю предлагается ввести туда имя. Затем появляется новое диалоговое окно с сообщением, которое содержит введенное пользователем имя. Но это, если имя пользователь ввел. Однако пользователь в первом окне может нажать кнопку отмены или закрыть окно с помощью системной пиктограммы с крестом. Вот такая ситуация и обрабатывается с помощью условного оператора. А именно, мы воспользуемся тем, что если пользователь закрывает окно с полем ввода без ввода текста, то соответствующий метод возвращает пустую текстовую строку. Чтобы легче было понять, как организован программный код, рассмотрим результат выполнения программы. Сначала, как указывалось, появляется окно с полем ввода, представленное на рис. 3.3.

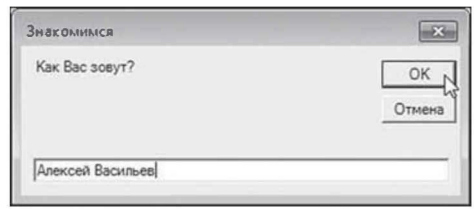


Рис. 3.3. Диалоговое окно с полем для ввода имени пользователя

Если мы вводим в поле ввода текст и щелкаем кнопку **ОК**, то следующим появляется окно с информационной пиктограммой (буква **i** в синем круге). Текст сообщения в окне содержит имя, введенное на предыдущем этапе. Окно имеет название **Знакомство состоялось**. Ситуация проиллюстрирована на рис. 3.4.

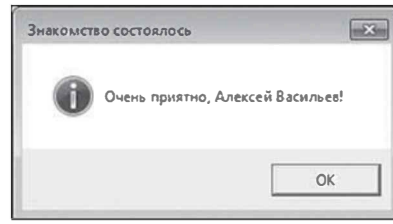


Рис. 3.4. Окно, отображаемое в случае, если пользователь ввел имя

Но если поле для ввода текста оставить пустым или вместо кнопки **ОК** в окне с полем ввода щелкнуть кнопку **Отмена** или системную пиктограмму (см. рис. 3.3), то появится окно как на рис. 3.5.

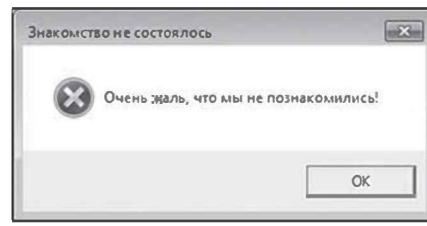


Рис. 3.5. Окно, отображаемое в случае, если пользователь не ввел имя

Окно содержит сообщение, пиктограмму ошибки (белый косой крест в красном круге) и имеет название **Знакомство не состоялось**. Подчеркнем, что появление такого окна — следствие выполнения программы, а не результат ошибки.

Теперь проанализируем программный код. Есть несколько важных моментов. Один из них связан с подходом, использованным в программе: на финальной стадии отображается окно с сообщением. Для отображения этого окна необходимо определить три параметра: текст сообщения, тип пиктограммы и название окна. Собственно, в процессе выполнения программы эти параметры определяются, и, после того как они определены, отображается окно. При определении указанных характеристик использован условный оператор.

В программе объявляется переменная `icon`, предназначенная для определения типа пиктограммы. Переменная объявлена как относящаяся к типу `MessageBoxIcon`. Напомним, что `MessageBoxIcon` — это перечисление. В этом перечислении есть константы, используемые для определения типа пиктограммы в диалоговом окне. Также мы используем три текстовые переменные `msg` (текст сообщения), `title` (заголовок окна), `name` (имя пользователя, которое вводится в поле ввода).

Для отображения окна с полем ввода используем метод `InputBox()` класса `Interaction`. Результат (введенное пользователем значение) записывается в переменную `name`. А дальше в игру вступает условный оператор. В условном операторе проверяется условие `name==""`. Это выражение возвращает значение `true`, если в переменную `name` записана пустая текстовая строка `""`. Это имеет место, если пользователь нажал кнопку **Отмена**, закрыл окно с полем ввода щелчком по системной пиктограмме или если пользователь щелкнул кнопку **ОК** при пустом поле ввода. В иных случаях значение переменной `name` будет иным и условие в условном операторе окажется ложным (значение `false`).

Если условие истинно, то командой `icon=MessageBoxIcon.Error` значение переменной `icon` определяется константой, соответствующей пиктограмме ошибки. Текст сообщения записывается в переменную `msg` командой `msg="Очень жаль, что мы не познакомились!"`. Наконец, командой `title="Знакомство не состоялось"` задается значение, используемое впоследствии в качестве названия для второго диалогового окна. Напомним, что все эти команды выполняются, если значением переменной `name` является пустая текстовая строка. В противном случае команды выполняются другие. Они собраны в `else`-блоке. Там командами `icon=MessageBoxIcon.Information`, `msg="Очень приятно, "+name+"!"` и `title="Знакомство состоялось"` определяются значения (но не такие, как в `if`-блоке) для переменных `icon`, `msg` и `title`. Пиктограмма в этом случае используется информационная, а текст сообщения содержит значение, которое указал пользователь в поле ввода.

После того как значения означенных переменных определены (а они в любом случае определяются, но происходит это по-разному в разных случаях), командой `MessageBox.Show(msg, title, MessageBoxButtons.OK, icon)` отображается диалоговое окно с сообщением.

Такую же программу (по функциональным возможностям) можно написать с использованием условного оператора в упрощенной форме. Соответствующий программный код представлен в листинге 3.2.

Листинг 3.2. Условный оператор в упрощенной форме

```
using System.Windows.Forms;
using Microsoft.VisualBasic;
class AnotherIfDemo{
   static void Main(){
      // Пиктограмма (сначала это пиктограмма ошибки):
      MessageBoxIcon icon=MessageBoxIcon.Error;
      // Текст сообщения:
      string msg="Очень жаль, что мы не познакомились!",
      // Заголовок окна:
      title="Знакомство не состоялось",
      // Переменная для записи имени пользователя:
      name;
      // Отображение окна с полем ввода:
      name=Interaction.InputBox(
         // Текст над полем ввода:
         "Как Вас зовут?",
         // Заголовок окна:
         "Знакомимся");
      // Условный оператор в упрощенной форме:
      if(name!=""){ // Если не пустая текстовая строка
         // Новый тип пиктограммы:
         icon=MessageBoxIcon.Information;
         // Новый текст сообщения:
         msg="Очень приятно, "+name+"!";
         // Новый заголовок окна:
         title="Знакомство состоялось";
      } // Завершение условного оператора
      // Отображение окна с сообщением:
```

```
        MessageBox.Show(msg, title, MessageBoxButtons.OK, icon);
    }
}
```

Данная программа функционирует точно так же, как и предыдущая программа. Но организация программного кода иная. Там переменные `icon`, `msg` и `title` сразу получают такие значения, как в случае, когда пользователь отказывается вводить имя. Другими словами, априори мы исходим из того, что пользователь не введет имя в поле первого диалогового окна. Но после того как определяется значение переменной `name`, выполняется проверка этого значения. Для проверки использован условный оператор в сокращенной форме. В условном операторе проверяется условие `name!=""`, истинное в случае, если в переменную `name` записана непустая текстовая строка. Если так, то значения переменных `icon`, `msg` и `title` переопределяются (переменным присваиваются новые значения). Если условие `name!=""` ложно, то переменные `icon`, `msg` и `title` остаются с теми значениями, которые им были присвоены в самом начале (при объявлении).

Вложенные условные операторы

> Ты свои КЦ им не показывай. И не думай о них. Ты мой КЦ покажи. И больше полспички пе давай, гравицапа полспички стоит.
>
> *из к/ф «Кин-дза-дза»*

Условные операторы могут быть вложенными — то есть в одном условном операторе (внешний оператор) может использоваться другой условный оператор (внутренний, или вложенный оператор), а в нем еще один условный оператор и так далее. Причем внутренние условные операторы могут использоваться и в `if`-блоке, и в `else`-блоке внешнего условного оператора. Например, ниже приведен шаблонный код для случая, когда внутренний условный оператор размещен внутри `if`-блок внешнего условного оператора (жирным шрифтом выделены ключевые элементы шаблона):

```
// Внешний условный оператор:
if(условие){
    // Внутренний условный оператор:
```

```
   if(условие){
      // Команды if-блока внутреннего оператора
   }
   // Блок else внутреннего оператора:
   else{
      // Команды else-блока внутреннего оператора
   }
}
// Блок else внешнего оператора:
else{
   // Команды else-блока внешнего оператора
}
```

Достаточно популярной является схема, когда внутренний условный оператор размещается в `else`-блоке внешнего оператора. Если таких вложений несколько, то получается конструкция, позволяющая последовательно проверять несколько условий. Ниже приведен подобный код для случая, когда используется четыре вложенных условных оператора (жирным шрифтом выделены ключевые элементы кода):

```
if(условие_1){
   // Команды выполняются, если условие_1 истинно
}
else if(условие_2){
   // Команды выполняются, если условие_2 истинно
}
else if(условие_3){
   // Команды выполняются, если условие_3 истинно
}
else if(условие_4){
   // Команды выполняются, если условие_4 истинно
}
else{
   // Команды выполняются, если все условия ложны
}
```

Хотя конструкция может выглядеть несколько необычно, но в действительности это всего лишь вложенные условные операторы. Просто нужно учесть несколько моментов. В первую очередь, следует обратить внимание, что `else`-блок каждого из условных операторов формируется всего одним оператором — внутренним условным. Поэтому `else`-блоки, за исключением последнего, описываются без использования фигурных скобок.

НА ЗАМЕТКУ

Напомним, что если `if`-блок или `else`-блок состоит из одной команды, то фигурные скобки можно не использовать. Условный оператор — это одна «составная» команда. Поэтому если `else`-блок или `if`-блок состоит из условного оператора, то его в фигурные скобки можно не заключать.

Условия в условных операторах проверяются последовательно: сначала проверяется условие во внешнем условном операторе (`условие_1`). Если условие истинно, то выполняются команды в `if`-блоке этого оператора. Если условие ложно, то выполняется условный оператор в `else`-блоке внешнего оператора — и проверяется условие во внутреннем операторе (`условие_2`). Если это условие истинно, то выполняются команды в `if`-блоке данного условного оператора. Если условие ложно, проверяется условие в следующем условном операторе (`условие_3`), и так далее. В итоге получается следующая схема: проверяется первое условие, и если оно ложно, проверяется второе условие. Если второе условие ложно, проверяется третье условие. Если оно ложно, проверяется четвертое условие, и так далее. Самый последний `else`-блок выполняется, если все условия оказались ложными.

В листинге 3.3 представлена программа, в которой используются вложенные условные операторы.

Листинг 3.3. Вложенные условные операторы

```
using System;
class NestedIfDemo{
   static void Main(){
      // Текстовая переменная:
      string txt;
      // Отображение сообщения:
```

```
      Console.Write("Введите текст: ");
      // Считывание текстовой строки:
      txt=Console.ReadLine();
      // Если введена непустая строка:
      if(txt!=""){
         // Отображение сообщения:
         Console.WriteLine("Спасибо, что ввели текст!");
         // Если в строке больше десяти символов:
         if(txt.Length>10){
            // Отображение сообщения:
            Console.WriteLine("Ого, как много букв!");
         }
         // Если в строке не больше десяти символов:
         else{
            // Отображение сообщения:
            Console.WriteLine("Ого, как мало букв!");
         }
      }
      // Если введена пустая строка:
      else{
         Console.WriteLine("Жаль, что не ввели текст!");
      }
   }
}
```

Алгоритм выполнения программы простой: пользователю предлагается ввести текстовое значение, и это значение считывается и записывается в текстовую переменную. Затем в дело вступают вложенные условные операторы. Во внешнем операторе проверяется, не пустая ли введенная пользователем текстовая строка. И если строка не пустая, то во внутреннем условном операторе проверяется, не превышает ли длина строки значение `10`. В зависимости от истинности или ложности этих условий в консольное окно выводятся разные сообщения.

Для записи текстового значения, введенного пользователем, используется переменная `txt`. Значение считывается командой `txt=Console.ReadLine()`. Во внешнем условном операторе проверяется условие `txt!=""`. Если условие ложно, то выполняется команда `Console.WriteLine("Жаль, что не ввели текст!")` в `else`-блоке внешнего условного оператора. Если же условие `txt!=""` во внешнем условном операторе истинное, то в `if`-блоке сначала выполняется команда `Console.WriteLine("Спасибо, что ввели текст!")`, после чего на сцену выходит внутренний условный оператор. В этом операторе проверяется условие `txt.Length>10`. Здесь мы воспользовались свойством `Length`, которое возвращает количество символов в текстовой строке. Следовательно, условие `txt.Length>10` истинно в случае, если текстовая строка, на которую ссылается переменная `txt`, содержит больше `10` символов. Если так, то выполняется команда `Console.WriteLine("Ого, как много букв!")` в `if`-блоке внутреннего условного оператора. Если условие `txt.Length>10` ложно, то выполняется команда `Console.WriteLine("Ого, как мало букв!")` в `else`-блоке внутреннего условного оператора.

Таким образом, возможны три принципиально разных случая:

- пользователь не ввел текст;
- пользователь ввел текст, состоящий больше чем из 10 символов;
- пользователь ввел текст, состоящий не больше чем из 10 символов.

Результат выполнения программы может быть таким (здесь и далее жирным шрифтом выделено значение, которое вводит пользователь).

Результат выполнения программы (из листинга 3.3)

```
Введите текст: Изучаем C#
Спасибо, что ввели текст!
Ого, как мало букв!
```

Это пример ситуации, когда введенный пользователем текст содержит не больше 10 символов. Если пользователь вводит текст, состоящий из более чем 10 символов, то результат может быть следующим.

Результат выполнения программы (из листинга 3.3)

```
Введите текст: Продолжаем изучать C#
```

```
Спасибо, что ввели текст!
Ого, как много букв!
```

Наконец, если пользователь не вводит текст, то результат выполнения программы такой.

Результат выполнения программы (из листинга 3.3)

```
Введите текст:
Жаль, что не ввели текст!
```

Еще один пример, рассматриваемый далее, иллюстрирует схему с вложенными условными операторами, когда каждый следующий внутренний условный оператор формирует `else`-блок внешнего условного оператора. Программа, представленная в листинге 3.4, выполняется по простой схеме: пользователь вводит целое число (предположительно, от 1 до 4), а программа выводит текстовое название этого числа.

Листинг 3.4. Определение числа

```
using System;
class AnotherNestedIfDemo{
   static void Main(){
      // Переменная для запоминания введенного числа:
      int number;
      // Отображение сообщения:
      Console.Write("Введите целое число: ");
      // Считывание числа:
      number=Int32.Parse(Console.ReadLine());
      // Если введена единица:
      if(number==1) Console.WriteLine("Единица");
      // Если введена двойка:
      else if(number==2) Console.WriteLine("Двойка");
      // Если введена тройка:
      else if(number==3) Console.WriteLine("Тройка");
      // Если введена четверка:
```

```
      else if(number==4) Console.WriteLine("Четверка");
      // Все прочие случаи:
      else Console.WriteLine("Неизвестное число");
   }
}
```

Программа «узнает» числа от 1 до 4 включительно. В начале выполнения программы командой `Console.Write("Введите целое число: ")` отображается приглашение ввести целое число. Число считывается командой `number=Int32.Parse(Console.ReadLine())` и записывается в переменную `number`. Затем задействуется блок из вложенных условных операторов, в каждом из которых проверяется введенное значение. При наличии совпадения в консольном окне появляется сообщение соответствующего содержания (название введенного числа). Если пользователь ввел число, не попадающее в диапазон значений от 1 до 4, то выполняется команда `Console.WriteLine("Неизвестное число")`. Ниже показано, как может выглядеть результат выполнения программы, если пользователь вводит «знакомое» для программы число (здесь и далее жирным шрифтом выделено введенное пользователем значение).

Результат выполнения программы (из листинга 3.4)

```
Введите целое число: 2
Двойка
```

Если программе не удается идентифицировать число, то результат такой.

Результат выполнения программы (из листинга 3.4)

```
Введите целое число: 5
Неизвестное число
```

Как мы увидим далее, вложенные условные операторы — далеко не единственный способ организовать проверку нескольких условий в программе.

Оператор выбора switch

Нет-нет, шуба подождет. Я считаю, что главное — посмотреть на мир.

из к/ф «Бриллиантовая рука»

Оператор выбора `switch` позволяет выполнить проверку значения некоторого выражения. В языке C# выражение, которое проверяется с помощью оператора выбора `switch`, может возвращать значение целочисленного, символьного или текстового типа. Описывается оператор выбора следующим образом. Сначала указывается ключевое слово `switch`, после которого в круглых скобках следует проверяемое выражение. Тело оператора выбора заключается в фигурные скобки. Там описываются `case`-блоки. Каждый такой блок начинается с ключевого слова `case`. После него указывается контрольное значение (заканчивается двоеточием). Каждый `case`-блок содержит набор команд, заканчивающийся инструкцией `break`. Контрольные значения в `case`-блоках сравниваются со значением выражения в `switch`-инструкции. Если совпадение найдено, то выполняются команды в соответствующем `case`-блоке. На случай, если совпадения нет, в операторе выбора `switch` может быть предусмотрен блок `default`. Общий шаблон описания оператора выбора `switch` с тремя `case`-блоками представлен ниже (жирным шрифтом выделены ключевые элементы шаблона):

```
switch(выражение){
   case значение_1:
      // Команды
      break;
   case значение_2:
      // Команды
      break;
   case значение_3:
      // Команды
      break;
   default:
      // Команды
      break;
}
```

Блок `default` не является обязательным — его можно не указывать совсем. На рис. 3.6 показана схема, иллюстрирующая способ выполнения оператора выбора `switch`.

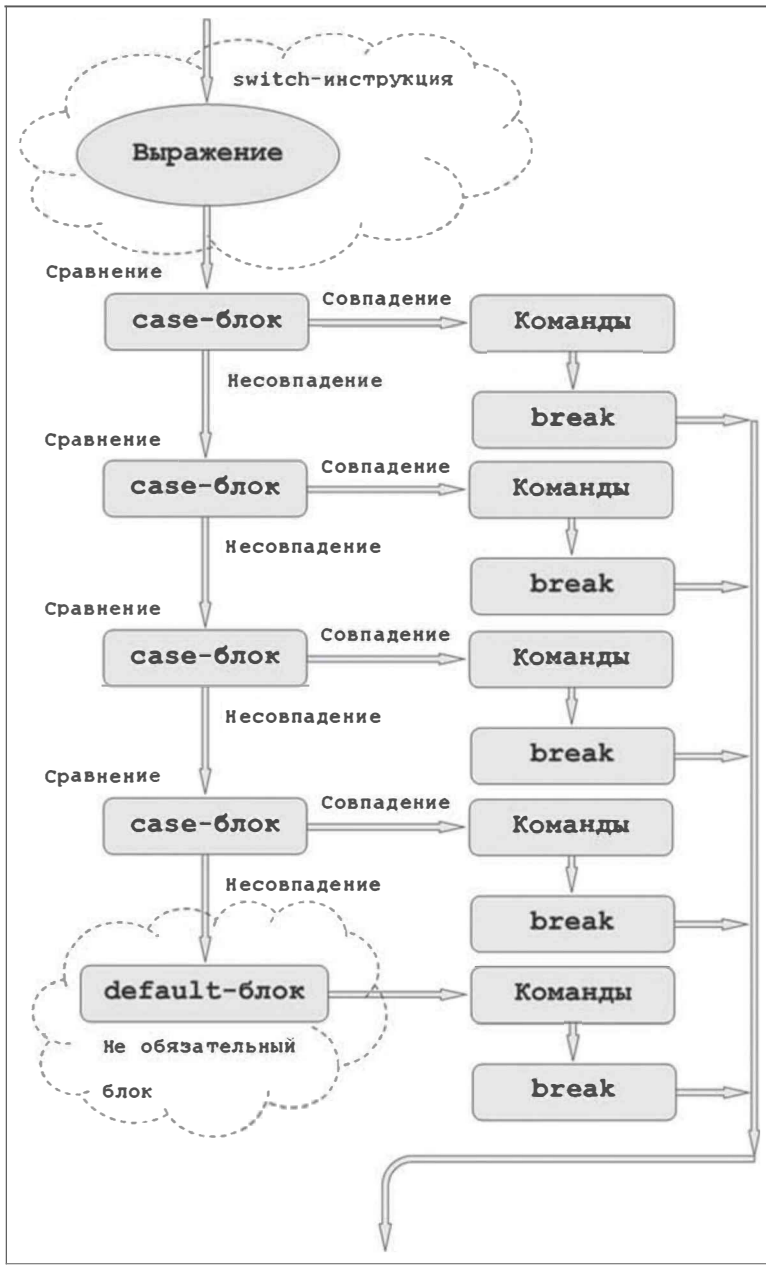


Рис. 3.6. Выполнение оператора выбора

Таким образом, при выполнении оператора выбора сначала вычисляется значение выражения в switch-инструкции. Полученное значение последовательно сравнивается с контрольными значениями в case-блоках. При первом же совпадении выполняются команды в соответствующем case-блоке — вплоть до инструкции break. Если совпадение не найдено и в операторе выбора есть default-блок, то выполняются команды в этом блоке. Если совпадение не найдено и default-блока в операторе выбора нет, то ничего не происходит.

ПОДРОБНОСТИ

И условный оператор, и оператор выбора, и рассматриваемые далее операторы цикла — все перечисленные управляющие инструкции есть не только в языке C#, но и, скажем, в языках C++ и Java. Даже больше — там они практически такие же, как в языке C#. И в плане синтаксиса, и в плане принципов выполнения. Но вот оператор выбора в C# немного особый. Во-первых, кроме целочисленных и символьных выражений в операторе выбора могут использоваться и текстовые выражения. Такого нет в языке C++ (там проверяемое выражение может быть целочисленным или символьным). Во-вторых, в C# в операторе выбора, если case-блок непустой, он должен заканчиваться break-инструкцией. В отличие от C#, в языках C++ и Java case-блок может заканчиваться инструкцией break, но это не обязательно. Вообще, инструкция break используется для завершения работы операторов цикла или оператора выбора. Само по себе завершение команд в case-блоке оператора выбора работу этого оператора не завершает. Другими словами, если в каком-то case-блоке команды выполнились, то это к автоматическому завершению выполнения всего оператора выбора не приводит. Чтобы завершить оператор выбора, нужна инструкция break. Чисто гипотетически, если бы case-блок не заканчивался инструкцией break, то после завершения выполнения команд в этом блоке должно было бы начаться выполнение команд в следующем case-блоке (даже если нет совпадения с контрольным значением в этом блоке). В языках C++ и Java все так и происходит. Что касается языка C#, то здесь при проверке значения выражения и сравнении его с контрольными значениями case-блоки могут перебираться не в том порядке, как они указаны в операторе выбора. Поэтому выполнение кода необходимо ограничить лишь блоком, для которого выявлено совпадение контрольного значения со значением проверяемого выражения. Как бы там ни было, но в языке C# непустые case-блоки и блок default заканчиваются инструкцией break. Как и зачем используются пустые case-блоки — обсудим немного позже.

Несложно заметить, что в некотором смысле оператор выбора напоминает блок из вложенных условных операторов, хотя полной аналогии здесь, конечно, нет. Небольшой пример, в котором использован оператор выбора, представлен в листинге 3.5. Фактически это альтернативный способ реализации программы с вложенными условными операторами, в которой по введенному пользователем числу определяется название этого числа (см. листинг 3.4). Но на этот раз вместо условных операторов использован оператор выбора `switch`.

Листинг 3.5. Знакомство с оператором выбора

```
using System;
using System.Windows.Forms;
using Microsoft.VisualBasic;
class SwitchDemo{
   static void Main(){
      // Переменная для запоминания введенного числа:
      int number;
      // Переменная для записи названия числа:
      string name;
      // Считывание числа:
      number=Int32.Parse(
         Interaction.InputBox(
            // Текст над полем ввода:
            "Введите число:",
            // Заголовок окна:
            "Число")
      );
      // Использование оператора выбора для определения
      // названия введенного числа:
      switch(number){
         case 1:              // Если ввели число 1
            name="Единица";   // Название числа
            break;            // Завершение блока
         case 2:              // Если ввели число 2
```

```
            name="Двойка";    // Название числа
            break;            // Завершение блока
         case 3:              // Если ввели число 3
            name="Тройка";    // Название числа
            break;            // Завершение блока
         case 4:              // Если ввели число 4
            name="Четверка";  // Название числа
            break;            // Завершение блока
         default:             // Ввели другое число
            // Текст сообщения:
            name="Неизвестное число";
            break;            // Завершение блока
      } // Завершение оператора выбора
      // Отображение сообщения:
      MessageBox.Show(name,"Число");
   }
}
```

Предыдущая версия программы была консольной. Эта версия для ввода числа и вывода сообщения использует диалоговые окна. Но это отличие «косметическое». Идеологически важное отличие состоит в том, что здесь мы используем оператор выбора `switch`. Целочисленная переменная `number` предназначена для записи числа, которое вводит пользователь. Текстовая переменная `name` нужна для того, чтобы сформировать и запомнить текстовое значение — название введенного пользователем числа. Для считывания числа используем статический метод `InputBox()` класса `Interaction`. Для преобразования текстового представления числа в целое число используем статический метод `Parse()` из структуры `Int32`. Результат записывается в переменную `number`.

Для проверки значения переменной `number` использован оператор выбора. Проверяемым выражением в нем является эта переменная. В `case`-блоках указаны контрольные значения — целые числа от `1` до `4` включительно (всего четыре `case`-блока). При наличии совпадения значения переменной `number` и контрольного значения в `case`-блоке

получает значение переменная `name` (значение переменной — название введенного числа). На случай, если совпадения не будет, предусмотрен `default`-блок. В этом блоке значением переменной `name` присваивается текст, сообщающий о невозможности идентифицировать число.

Таким образом, было или не было совпадение, переменная `name` получит значение. Это значение используется в команде `MessageBox.Show(name,"Число")`, которой отображается диалоговое окно с сообщением (название числа или сообщение о том, что число неизвестно).

На рис. 3.7 показано окно с полем ввода, в которое введено число `2`.

Рис. 3.7. В поле введено число 2

После подтверждения ввода появляется новое диалоговое окно, показанное на рис. 3.8.

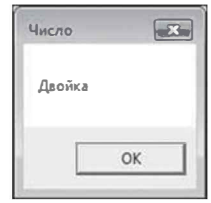


Рис. 3.8. Окно с сообщением после ввода числа 2

На рис. 3.9 показано окно с полем ввода, в котором пользователь указал число `5`.

Рис. 3.9. В поле введено число 5

После щелчка по кнопке **ОК** появляется окно с сообщением о том, что число неизвестно. Окно показано на рис. 3.10.

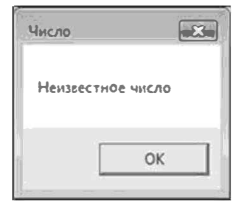


Рис. 3.10. Окно с сообщением после ввода числа 5

Стоит заметить, что если пользователь вместо ввода числа в окне с полем щелкнет, например, кнопку **Отмена**, то возникнет ошибка. Желающие могут подумать, как следует модифицировать программный код, чтобы подобная ситуация обрабатывалась корректно.

НА ЗАМЕТКУ

В программе использовано несколько пространств имен. Пространство имен `System.Windows.Forms` необходимо подключить для использования статического метода `Show()` из класса `MessageBox`. Метод отображает диалоговое окно с сообщением (в данном случае название введенного пользователем числа). Пространство имен `Microsoft.VisualBasic` нужно для использования статического метода `InputBox()` из класса `Interaction`. Метод отображает окно с полем ввода. Напомним, что в данном случае мы прибегаем к помощи средств разработки языка Visual Basic. Наконец, пространство имен `System` нам понадобилось, поскольку мы используем статический метод `Parse()` из структуры `Int32`. Вообще-то мы еще используем и текстовый тип `String`. Это класс из пространства имен `System`. Но в программе ссылка на текстовый тип выполнена через идентификатор `string`, являющийся псевдонимом для инструкции `System.String`. Так что `string` можно было бы использовать и без подключения пространства имен `System`.

Как отмечалось ранее, `case`-блоки в операторе выбора могут быть пустыми. Так поступают в случае, если необходимо, чтобы для нескольких контрольных значений выполнялись одни и те же команды. Если так, то вместо дублирования команд в соответствующих `case`-блоках (что неудобно и не очень рационально) используют несколько идущих один за другим пустых блоков. Причем в таких пустых `case`-блоках нет

даже `break`-инструкций. Пример подобного подхода проиллюстрирован в следующей программе, представленной в листинге 3.6.

Листинг 3.6. Оператор выбора с пустыми case-блоками

```
using System;
using System.Windows.Forms;
using Microsoft.VisualBasic;
class AnotherSwitchDemo{
   static void Main(){
      // Переменная для запоминания введенного числа:
      int number;
      // Переменная для текста сообщения:
      string txt="";   // Начальное значение переменной
      // Считывание числа:
      number=Int32.Parse(
         Interaction.InputBox(
            // Текст над полем ввода:
            "Введите целое число от 1 до 9:",
            // Заголовок окна:
            "Число")
      );
      // Проверка значения переменной number:
      switch(number){  // Оператор выбора
         case 1:   // Если значение 1
         case 9:   // Если значение 9
            // Текст сообщения:
            txt="Вы ввели нечетное,\n но не простое число.";
            break; // Завершение блока
         case 2:   // Если значение 2
         case 3:   // Если значение 3
         case 5:   // Если значение 5
         case 7:   // Если значение 7
```

```
            // Текст сообщения:
            txt="Вы ввели простое число.";
            break; // Завершение блока
         case 4:   // Если значение 4
         case 8:   // Если значение 8
            // Текст сообщения:
            txt="Вы ввели число – степень двойки.";
            break; // Завершение блока
         case 6:   // Если значение 6
            // Текст сообщения:
            txt="Вы ввели 6 – совершенное число.";
            break; // Завершение блока
      }
      // Отображение диалогового окна с сообщением:
      MessageBox.Show(txt,"Число");
   }
}
```

Данная программа в некотором смысле напоминает предыдущую. Сначала пользователю предлагается ввести целое число (в диапазоне от 1 до 9). Результат записывается в переменную `number`. После этого с помощью оператора выбора проверяется значение этой переменной. Мы «классифицируем» числа по следующим категориям:

- Простые числа — числа, которые не имеют других делителей, кроме единицы и себя самого. В диапазоне чисел от 1 до 9 простыми являются числа 2, 3, 5 и 7.
- Числа, которые являются степенью двойки — в диапазоне от 1 до 9 в эту категорию попадают числа 4 ($2^2 = 4$) и 8 ($2^3 = 8$).
- Число 6 является совершенным — сумма его делителей (числа 1, 2 и 3) равна самому этому числу. Следующее совершенное число — это 28 (его делители 1, 2, 4, 7 и 14 в сумме дают 28), но оно не попадает в интервал от 1 до 9.
- Нечетные числа, которые при этом не являются простыми — в указанном диапазоне это числа 1 и 9.

Введенное пользователем число с помощью оператора выбора «приписывается» к одной из перечисленных групп. И в этом операторе выбора мы используем пустые case-блоки. Например, есть в операторе выбора такая конструкция (комментарии для удобства удалены):

```
case 2:
case 3:
case 5:
case 7:
   txt="Вы ввели простое число.";
   break;
```

Как это работает? Допустим, пользователь ввел значение 2. При сравнении этого значения с контрольными значениями в case-блоке совпадение имеет место для блока case 2. Поэтому будут выполнены все команды от места, где идентифицировано совпадение, до первой инструкции break. В результате выполняется команда txt="Вы ввели простое число.". Теперь предположим, что пользователь ввел значение 5. Совпадение проверяемого (переменная number) и контрольного значения имеет место в блоке case 5. Он пустой. Но это все равно case-блок. Поэтому выполняются команды от места совпадения до первой инструкции break. Итог такой — выполняется команда txt="Вы ввели простое число.". То же получаем, когда пользователь вводит значение 3 и 7. Прочие блоки в операторе выбора работают по такому же принципу.

Внешне все выглядит следующим образом. При запуске программы на выполнение появляется окно с полем ввода, в которое следует ввести число от 1 до 9. Окно показано на рис. 3.11.

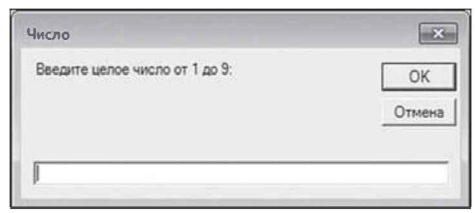


Рис. 3.11. Окно с полем для ввода числа в диапазоне от 1 до 9

Если пользователь вводит числа 1 или 9, то следующим появляется окно, представленное на рис. 3.12.

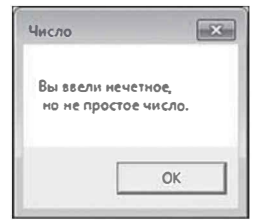


Рис. 3.12. Окно появляется, если пользователь вводит число 1 или 9

НА ЗАМЕТКУ

В операторе выбора в одном из `case`-блоков (для значений `1` и `9`) при определении текстового значения переменной `txt` в текстовом литерале использована инструкция `\n`. Напомним, что это инструкция перехода к новой строке. При отображении соответствующего текста в том месте, где размещена инструкция `\n`, выполняется переход к новой строке. Результат можно видеть на рис. 3.12, где текст в диалоговом окне отображается в две строки.

В случае, если пользователь вводит число `2`, `3`, `5` или `7`, появится окно, представленное на рис. 3.13.

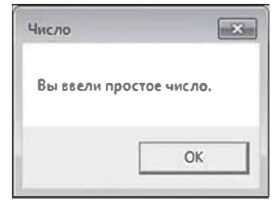


Рис. 3.13. Окно появляется, если пользователь вводит число 2, 3, 5 или 7

Если пользователь вводит число `6`, появляется окно, показанное на рис. 3.14.

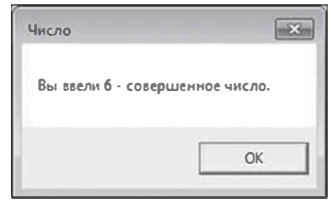


Рис. 3.14. Окно появляется, если пользователь вводит число 6

Наконец, если пользователь вводит число 4 или 8, то появляется окно, показанное на рис. 3.15.

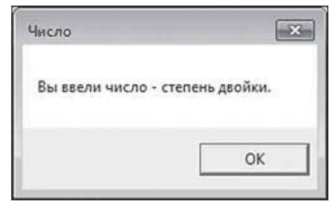


Рис. 3.15. Окно появляется, если пользователь вводит числа 4 или 8

Хочется обратить внимание на несколько обстоятельств. Во-первых, в операторе выбора мы не использовали `default`-блок. Как упоминалось ранее, этот блок не является обязательным, чем мы, собственно, и воспользовались. Но поскольку теперь в `switch`-операторе `default`-блока нет, то теоретически может получиться, что в операторе переменной `txt` значение не будет присвоено — например, если пользователь введет число, которое не попадает в диапазон значений от 1 до 9. И это проблема, поскольку переменная `txt` используется в команде `MessageBox.Show(txt,"Число")` после оператора выбора. Такие ситуации компилятором отслеживаются, и выдается ошибка еще на этапе компиляции. Чтобы ее избежать, при объявлении переменной `txt` мы ей сразу присвоили пустое текстовое значение. В таком случае, даже если пользователь введет значение вне рекомендуемого диапазона и в операторе выбора значение переменной `txt` присвоено не будет, переменная останется со своим старым значением. Возможно, это не самый лучший способ решения проблемы, но, во всяком случае, при компиляции не будет ошибки.

Также стоит заметить, что, если пользователь в поле ввода введет не число, отменит ввод (щелкнув кнопку **Отмена**) или закроет окно щелчком по системной пиктограмме, возникнет ошибка. Чтобы ее избежать, можно воспользоваться системой перехвата и обработки исключений. Как это делается, кратко будет показано в одном из разделов в конце главы. Обработке исключений также посвящена одна из глав во второй части книги.

Оператор цикла while

Замечательная идея! Что ж она мне самому в голову не пришла?

из к/ф «Ирония судьбы, или С легким паром»

Оператор цикла позволяет многократно выполнять определенный набор команд. В языке C# существует несколько операторов цикла, и со всеми ними мы познакомимся. Но начнем с *оператора цикла* `while`. У него достаточно простой синтаксис. Описание оператора начинается с ключевого слова `while`. В круглых скобках после ключевого слова `while` указывается некоторое условие (выражение со значением логического типа). Затем в фигурных скобках указывается блок из команд, формирующих тело оператора цикла. Общий синтаксис оператора цикла `while`, таким образом, следующий (жирным шрифтом выделены ключевые элементы шаблона):

```
while(условие){
   // Команды
}
```

Выполняется оператор цикла `while` так. Сначала проверяется условие (в круглых скобках после ключевого слова `while`). Если условие истинно (значение `true`), то выполняются команды в теле оператора цикла. После этого снова проверяется условие. Если оно истинно, то снова выполняются команды в теле оператора цикла, после чего опять проверяется условие, и так далее. Оператор цикла выполняется до тех пор, пока при проверке условия оно не окажется ложным (значение `false`). Если при проверке условия оно оказывается ложным, команды в теле оператора цикла не выполняются, работа оператора цикла завершается и управление передается следующей инструкции после оператора цикла. Схема выполнения оператора цикла `while` проиллюстрирована на рис. 3.16.

Стоит заметить, что команды в теле оператора выполняются блоком — то есть условие в очередной раз проверяется только после того, как выполнены все команды в теле оператора цикла. Условие, указанное в круглых скобках после ключевого слова `while`, должно быть таким, чтобы при выполнении команд в теле оператора цикла оно в принципе могло измениться. Проще говоря, чтобы в операторе цикла начали выполняться команды, условие в самом начале должно быть равно `true`. А чтобы оператор цикла в какой-то момент завершил выполнение, условие должно стать равным `false`. Иначе получим бесконечный цикл.

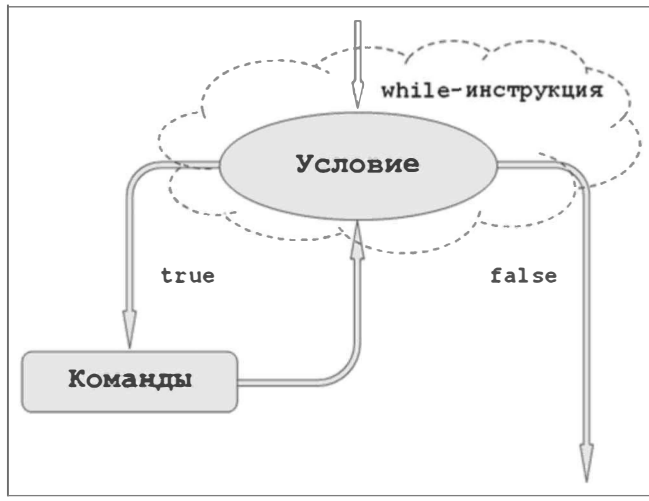


Рис. 3.16. Выполнение оператора цикла *while*

НА ЗАМЕТКУ

Инструкция `break`, с которой мы встретились при знакомстве с оператором выбора `switch`, позволяет завершать выполнение оператора цикла. Если в теле оператора цикла выполняется инструкция `break`, то оператор цикла прекращает свое выполнение вне зависимости от того, какое условие. Инструкция `continue` может использоваться в теле оператора цикла для досрочного завершения текущего цикла (итерации).

Если в теле оператора цикла всего одна команда, то фигурные скобки можно не использовать.

Далее мы рассмотрим небольшой пример, в котором используется оператор цикла `while`. Программа очень простая — в ней вычисляется сумма первых n нечетных чисел $1 + 3 + 5 + ... + (2n - 1)$ (значение параметра n в программе определяется с помощью переменной).

ПОДРОБНОСТИ

Для проверки результатов вычислений можно воспользоваться соотношением $1 + 3 + 5 + ... + (2n - 1) = n^2$. Поэтому, например, если вычисляется сумма первых 10 нечетных чисел, то такая сумма равна 100, то есть $1 + 3 + 5 + ... + 19 = 100$. Но в программе эти формулы не используются — вычисление суммы выполняется непосредственным суммированием слагаемых.

Интересующий нас программный код представлен в листинге 3.7.

Листинг 3.7. Использование оператора цикла while

```
using System;
class WhileDemo{
   static void Main(){
      // Количество слагаемых в сумме, индексная
      // переменная и значение суммы:
      int n=10, k=1, s=0;
      // Отображение сообщения:
      Console.Write("Сумма 1 + 3 + 5 +…+ {0} = ",2*n-1);
      // Оператор цикла:
      while(k<=n){
         s+=2*k-1;   // Добавляем слагаемое в сумму
         k++;        // Новое значение индексной переменной
      }
      // Отображение вычисленного значения:
      Console.WriteLine(s);
   }
}
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 3.7)

```
Сумма 1 + 3 + 5 +…+ 19 = 100
```

В программе мы объявляем три целочисленные переменные. Переменная `n` со значением `10` определяет количество слагаемых в сумме. Переменная `k` с начальным значением `1` нам нужна для подсчета количества слагаемых, добавленных к значению суммы. Эту переменную будем называть индексной. Переменная `s` с начальным значением `0` нам понадобится для записи в эту переменную значения суммы. Перед началом вычислений командой `Console.Write("Сумма 1 + 3 + 5 +…+ {0} = ",2*n-1)` в консольном окне выводится текстовое сообщение, формально обозначающее вычисляемую сумму. Текстовое значение

заканчивается знаком равенства, и переход к новой строке не выполняется (поскольку использован метод `Write()`).

ПОДРОБНОСТИ

Команда `Console.Write("Сумма 1 + 3 + 5 +…+ {0} = ",2*n-1)` выполняется так. Отображается текстовое значение `"Сумма 1 + 3 + 5 +…+ {0} = "`, но только вместо инструкции `{0}` подставляется значение выражения `2*n-1`, которое указано аргументом метода `Write()` после текстовой строки.

Здесь мы учли, что если вычисляется сумма n нечетных чисел, то последнее слагаемое в такой сумме равно $2n - 1$.

Для вычисления суммы использован оператор цикла `while`. В операторе цикла указано условие `k<=n` — то есть оператор цикла будет выполняться до тех пор, пока значение переменной `k` не превышает значения переменной `n`. Поскольку начальное значение переменной `k` равно `1`, а значение переменной `n` равно `10`, то при первой проверке условия `k<=n` его значение равно `true` (условие истинно). Поэтому начинается выполнение команд в теле оператора цикла. Там их всего две. Сначала командой `s+=2*k-1` к текущему значению суммы прибавляется очередное слагаемое, после чего командой `k++` значение индексной переменной увеличивается на единицу.

ПОДРОБНОСТИ

Мы приняли в расчет, что если k — это порядковый номер нечетного числа, то значение этого числа может быть вычислено по формуле $2k - 1$. Поэтому на каждой итерации к переменной `s`, в которую записывается значение суммы нечетных чисел, прибавляется значение `2*k-1`. Чтобы получить значение очередного нечетного числа, можно воспользоваться той же формулой, но только значение переменной `k` нужно увеличить на единицу.

После выполнения команд в теле цикла снова проверяется условие `k<=n`. Для использованных начальных значений переменных при второй проверке условие опять окажется истинным. Будут выполнены команды в теле оператора, потом снова начнется проверка условия. И так далее. Последний раз условие `k<=n` окажется истинным, когда значение переменной `k` будет равно значению переменной `n`. На этой итерации к значению переменной `s` прибавляется значение `2*n-1`, а значение

переменной k станет больше на единицу значения переменной n. В результате при проверке условия k<=n оно окажется ложным, и оператор цикла завершит свою работу.

По завершении оператора цикла в переменную s записана искомая сумма нечетных чисел. Поэтому командой Console.WriteLine(s) вычисленное значение отображается в консольном окне.

НА ЗАМЕТКУ

Если в рассмотренном выше программном коде переменной n вместо значения 10 присвоить значение 0, то для суммы нечетных чисел программа выдаст значение 0. Объяснение простое: при первой же проверке условия k<=n в операторе цикла, поскольку начальное значение переменной k равно 1, а значение переменной n равно 0, условие будет ложным. Поэтому команды в теле оператора цикла выполняться не будут. Следовательно, переменная s останется со своим начальным значением 0. Оно и будет отображено в консольном окне как результат вычислений.

Мы могли реализовать программу для вычисления суммы нечетных чисел немного иначе. Как минимум мы могли бы обойтись без индексной переменной k. Программа с «уменьшенным» количеством переменных приведена в листинге 3.8.

Листинг 3.8. Еще один пример использования оператора цикла while

```
using System;
class AnotherWhileDemo{
   static void Main(){
      // Количество слагаемых в сумме и значение суммы:
      int n=10, s=0;
      // Отображение сообщения:
      Console.Write("Сумма 1 + 3 + 5 +…+ {0} = ",2*n-1);
      // Оператор цикла:
      while(n>0){
         s+=2*n-1;   // Добавляем слагаемое в сумму
         n--;        // Новое значение переменной n
      }
```

```
        // Отображение вычисленного значения:
        Console.WriteLine(s);
    }
}
```

Результат выполнения этой программы точно такой же, как и в предыдущем случае (см. листинг 3.7). Но код немного изменился. Как отмечалось выше, мы больше не используем переменную `k`. В некотором смысле ее «роль» играет переменная `n`. Теперь в операторе цикла проверяется условие `n>0`, а в теле оператора цикла выполняются команды `s+=2*n-1` и `n--`. Фактически здесь вычисляется сумма $(2n - 1) + (2n - 3) + ... + 5 + 3 + 1$, которая, очевидно, равна сумме $1 + 3 + 5 + ... + (2n - 3) + (2n - 1)$. Другими словами, мы вычисляем ту же сумму, но слагаемые добавляем в обратном порядке. При начальном значении переменной `n` выражение `2*n-1` дает значение последнего слагаемого в сумме. После уменьшения значения переменной `n` на единицу выражение `2*n-1` дает значение предпоследнего слагаемого в сумме, и так далее, пока при единичном значении переменной `n` в соответствии с выражением `2*n-1` не получим значение `1` (первое слагаемое в сумме).

НА ЗАМЕТКУ

К моменту завершения оператора цикла значение переменной `n` равно `0`. Таким образом, начальное значение переменной `n` «теряется». В версии программы с индексной переменной (см. листинг 3.7) значение переменной `n` оставалось неизменным.

Оператор цикла do-while

> Это в твоих руках все горит, а в его руках все работает!
>
> *из к/ф «Покровские ворота»*

Синтаксис *оператора цикла* `do-while` напоминает синтаксис оператора цикла `while`. Описание начинается с ключевого слова `do`, после которого в фигурных скобках указываются команды, формирующие тело оператора цикла. Затем после блока из фигурных скобок следует ключевое слово `while`, и в круглых скобках — условие, истинность которого

является критерием для продолжения работы оператора цикла. Шаблон описания оператора цикла `do-while` представлен ниже (жирным шрифтом выделены ключевые элементы шаблона):

```
do{
   // Команды
}while(условие);
```

Выполняется оператор цикла `do-while` следующим образом. Сначала выполняются команды в теле оператора цикла. После их выполнения проверяется условие, указанное после ключевого слова `while`. Если условие окажется истинным, будут выполнены команды в теле оператора цикла и проверено условие. Процесс завершается, когда при проверке условия оно окажется ложным. Схема выполнения оператора цикла `do-while` проиллюстрирована на рис. 3.17.

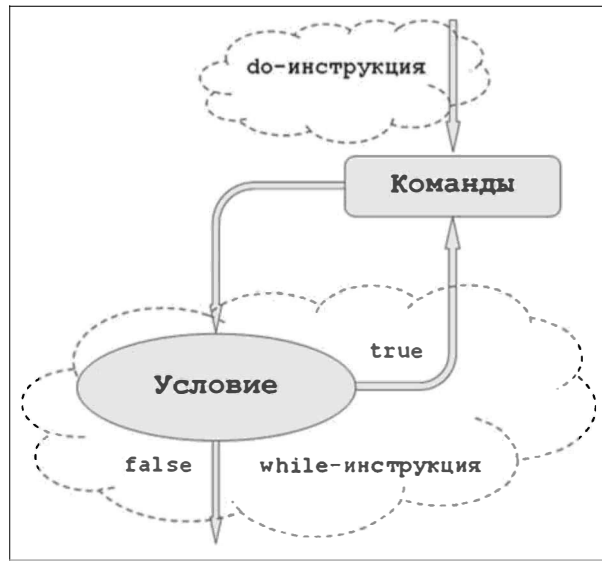


Рис. 3.17. Выполнение оператора `do-while`

Если сравнивать оператор цикла `do-while` с оператором цикла `while`, то легко заметить главное отличие этих операторов. Выполнение оператора `while` начинается с проверки условия, и если условие истинно, то выполняются команды в теле оператора цикла. В операторе цикла `do-while` сначала выполняются команды в теле оператора, и только после этого проверяется условие. Получается, что в операторе цикла

do-while команды из тела оператора будут выполнены по крайней мере один раз. С оператором while дела обстоят иначе: если при первой проверке условия оно окажется ложным, то команды в теле оператора вообще выполняться не будут.

Как пример использования оператора цикла do-while мы рассмотрим программу вычисления суммы нечетных чисел — то есть программу, аналогичную той, что реализовалась с использованием оператора цикла while (см. листинг 3.7). Рассмотрим программный код, представленный в листинге 3.9.

Листинг 3.9. Использование оператора цикла do-while

```
using System;
class DoWhileDemo{
   static void Main(){
      // Количество слагаемых в сумме, индексная
      // переменная и значение суммы:
      int n=10, k=1, s=0;
      // Отображение сообщения:
      Console.Write("Сумма 1 + 3 + 5 + … + {0} = ",2*n-1);
      // Оператор цикла:
      do{
         s+=2*k-1;   // Добавляем слагаемое в сумму
         k++;        // Новое значение индексной переменной
      }while(k<=n);
      // Отображение вычисленного значения:
      Console.WriteLine(s);
   }
}
```

Результат выполнения программы точно такой же, как и для случая, когда использовался оператор цикла while.

Результат выполнения программы (из листинга 3.9)

```
Сумма 1 + 3 + 5 + … + 19 = 100
```

Алгоритм выполнения программы по сравнению с исходной версией (см. листинг 3.7) практически не изменился. Но отличие все же есть. В программе из листинга 3.7 (использован оператор цикла `while`) сначала проверяется условие `k<=n`, а затем выполняются команды в теле цикла. В программе из листинга 3.9 (использован оператор `do-while`) сначала выполняются команды в теле цикла, а затем проверяется условие `k<=n`. Если значение переменной `n` не меньше `1`, то разницы, в общем-то, нет. Но если значение переменной `n` меньше `1`, то программы будут выдавать разные результаты. Например, при нулевом значении переменной `n` первая программа выдает значение `0` для суммы чисел, а вторая выдает значение `1`. Причина в том, что при первой же проверке условия `k<=n` оно оказывается ложным. Но в случае с оператором цикла `do-while` перед проверкой условия один раз будут выполнены команды в теле оператора цикла. Поэтому значение переменной `s` станет равным `1`.

Оператор цикла for

> — Один крокодил. Два крокодила. Три крокодила...
> — Бредит, бедняга. Похоже на тропическую лихорадку!
>
> *из м/ф «Приключения капитана Врунгеля»*

Синтаксис *оператора цикла* `for` немного сложнее по сравнению с тем, как описываются операторы цикла `while` и `do-while`. Описание оператора цикла начинается с ключевого слова `for`. В круглых скобках после ключевого слова размещается три блока инструкций. Блоки инструкций разделяются точкой с запятой, а инструкции внутри блока — запятыми. Затем указывается блок команд в фигурных скобках. Они формируют тело оператора цикла.

> **НА ЗАМЕТКУ**
>
> Если тело оператора цикла содержит всего одну команду, то фигурные скобки можно не использовать.

Шаблон описания оператора цикла `for` представлен ниже (жирный шрифт выделяет основные элементы шаблона):

```
for(первый блок; второй блок; третий блок){
   // Команды
}
```

Выполняется оператор цикла `for` следующим образом. Сначала выполняются команды в первом блоке (в круглых скобках после ключевого слова `for`). Эти команды выполняются один и только один раз в самом начале работы оператора цикла. Поэтому в первый блок обычно помещают команды, имеющие отношение к инициализации переменных, а сам блок нередко называют блоком *инициализации*. Во втором блоке размещается условие, поэтому второй блок обычно называют блоком *условия*. Это условие проверяется сразу после выполнения команд в первом блоке. Если условие ложно, то на этом выполнение оператора цикла заканчивается. Если условие истинно, то выполняются команды в теле оператора цикла. После этого выполняются команды в третьем блоке. Затем проверяется условие во втором блоке. Если условие ложно, работа оператора цикла прекращается. Если условие истинно, то выполняются команды в теле оператора цикла, затем в третьем блоке и снова проверяется условие. И так далее. Схема выполнения оператора цикла `for` проиллюстрирована на рис. 3.18.

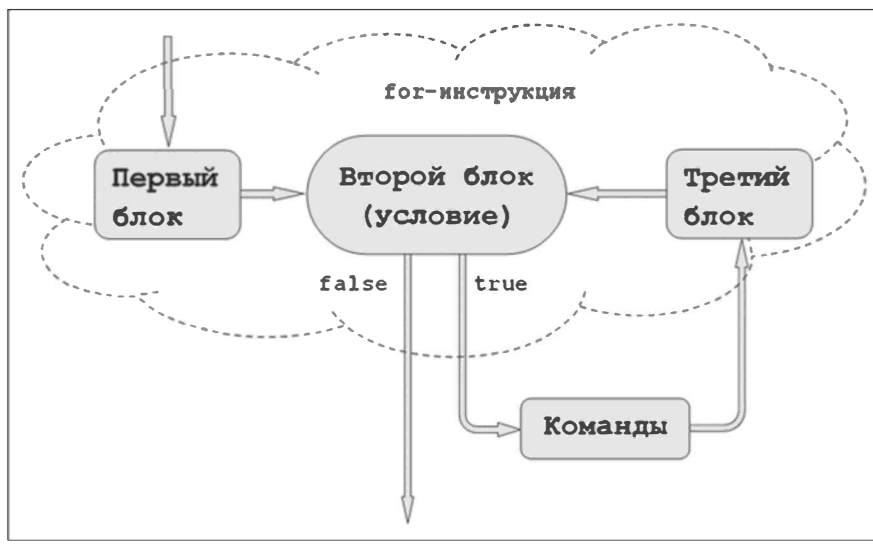


Рис. 3.18. Выполнение оператора цикла `for`

НА ЗАМЕТКУ

В третьем блоке обычно размещается команда, которой изменяется значение индексной переменной. Поэтому третий блок иногда называют блоком инкремента/декремента.

Имеет смысл обратить внимание читателя на несколько обстоятельств. Во-первых, легко заметить, что разница между командами в теле оператора цикла и в третьем блоке достаточно условная. Действительно, сначала выполняются команды в теле оператора цикла, а затем команды в третьем блоке. Поэтому имеет значение последовательность команд, но не то, в каком месте они находятся (в теле оператора или в третьем блоке). Во-вторых, блоки могут содержать по несколько команд (в таких случаях команды разделяются запятыми), а могут быть пустыми. Пустым может быть и второй блок. В этом случае он эквивалентен блоку с истинным условием — то есть фактически мы имеем дело с бесконечным циклом. Выход из такого бесконечного цикла может быть связан с использованием условного оператора и инструкции `break` (пример рассматривается чуть позже). Наконец, первый блок может содержать не просто команды присваивания значений переменным, а команды, объявляющие переменную или переменные. С одной стороны, удобно. С другой стороны, такие переменные доступны только в пределах оператора цикла. Далее мы рассмотрим небольшие примеры, иллюстрирующие работу оператора цикла `for`. Сначала рассмотрим программу в листинге 3.10. Эта программа, как несложно догадаться, вычисляет сумму нечетных чисел, но на этот раз используется оператор цикла `for`.

Листинг 3.10. Использование оператора цикла for

```
using System;
class ForDemo{
   static void Main(){
      // Количество слагаемых в сумме, индексная
      // переменная и значение суммы:
      int n=10, k, s=0;
      // Отображение сообщения:
      Console.Write("Сумма 1 + 3 + 5 +…+ {0} = ",2*n-1);
      // Оператор цикла:
      for(k=1; k<=n; k++){
         s+=2*k-1;   // Добавляем слагаемое в сумму
      }
      // Отображение вычисленного значения:
```

```
      Console.WriteLine(s);
   }
}
```

Результат выполнения программы таков.

Результат выполнения программы (из листинга 3.10)

```
Сумма 1 + 3 + 5 +…+ 19 = 100
```

Несложно заметить, что результат такой же, как и в рассмотренных ранее случаях. Что нового в программном коде? Как и ранее, объявляются три целочисленные переменные (`n` со значением `10`, `k` и `s` со значением `0`). Но на этот раз переменной `k` значение при объявлении не присваивается. Переменная `k` получает единичное значение в начале выполнения оператора цикла `for`, когда выполняется команда `k=1` в первом блоке. Эта команда выполняется только один раз в самом начале работы оператора цикла. После ее выполнения проверяется условие `k<=n` во втором блоке. При тех значениях, которые присвоены переменным `k` и `n`, условие истинно. Поэтому выполняется команда `s+=2*k-1` в теле оператора цикла и команда `k++` в третьем блоке. Затем снова проверяется условие. То есть, несмотря на изменившийся синтаксис программного кода, последовательность выполнения команд, по сути, осталась прежней. Отсюда и результат.

Еще одна небольшая вариация программы с оператором цикла `for` представлена в листинге 3.11 (для сокращения объема программного кода непринципиальные комментарии удалены из программы).

Листинг 3.11. Второй пример использования оператора цикла for

```
using System;
class ForDemoTwo{
   static void Main(){
      int n=10, s=0;
      Console.Write("Сумма 1 + 3 + 5 +…+ {0} = ",2*n-1);
      for(int k=1; k<=n; k++){
         s+=2*k-1;
      }
      Console.WriteLine(s);
   }
}
```

Результат выполнения программы такой же, как и в предыдущем случае. Особенность программного кода в том, что переменная k объявляется и инициализируется в первом блоке в операторе цикла for. Как следствие, такая переменная доступна только в операторе цикла for, но не за его пределами. Проще говоря, если бы мы захотели проверить значение переменной k после завершения выполнения оператора цикла, то на этапе компиляции возникла бы ошибка.

Другой способ организации оператора цикла, когда блоки в for-инструкции содержат по несколько команд, представлен в программе в листинге 3.12.

Листинг 3.12. Третий пример использования оператора цикла for

```
using System;
class ForDemoThree{
   static void Main(){
      int n=10, k, s;
      Console.Write("Сумма 1 + 3 + 5 +…+ {0} = ",2*n-1);
      for(k=1, s=0; k<=n; s+=2*k-1, k++);
      Console.WriteLine(s);
   }
}
```

В данном случае переменные n, k и s объявляются в главном методе программы, но начальные значения переменным k и s присваиваются в операторе цикла (в первом блоке). Команда s+=2*k-1 из тела оператора цикла перенесена в третий блок, и теперь там две команды (кроме упомянутой команды s+=2*k-1 там еще есть инструкция k++). В результате тело оператора цикла оказалось пустым, поэтому сразу после for-инструкции стоит точка с запятой.

ПОДРОБНОСТИ

Если тело оператора цикла пустое, то мы можем либо поставить точку с запятой после for-инструкции, либо поставить пустые фигурные скобки. Если ничего этого не сделать, то к оператору цикла будет относиться следующая (после оператора цикла) команда в программе — в данном случае это была бы команда Console.WriteLine(s).

Противоположная к предыдущей ситуация реализована в программном коде в листинге 3.13 — там два блока (первый и третий) в `for`-инструкции оператора цикла пустые.

Листинг 3.13. Четвертый пример использования оператора цикла for

```
using System;
class ForDemoFour{
   static void Main(){
      int n=10, k=1, s=0;
      Console.Write("Сумма 1 + 3 + 5 +…+ {0} = ",2*n-1);
      for(; k<=n;){
         s+=2*k-1;
         k++;
      }
      Console.WriteLine(s);
   }
}
```

Здесь переменные `k` и `s` получают начальные значения при объявлении, поэтому в первом блоке нет необходимости выполнять присваивание значений переменным. Что касается третьего блока, то команды из него вынесены в тело оператора цикла: там теперь есть инструкции `s+=2*k-1` и `k++`. Поэтому третий блок тоже пустой.

Наконец, совсем «экзотический» случай представлен в программном коде в листинге 3.14. В этой программе в операторе цикла `for` все три блока (включая второй блок с условием) являются пустыми.

Листинг 3.14. Пятый пример использования оператора цикла for

```
using System;
class ForDemoFive{
   static void Main(){
      int n=10, k=1, s=0;
      Console.Write("Сумма 1 + 3 + 5 +…+ {0} = ",2*n-1);
      for(;;){           // Все блоки пустые
         s+=2*k-1;
```

```
            k++;
            // Условный оператор:
            if(k>n) break;   // Завершение оператора цикла
        }
        Console.WriteLine(s);
    }
}
```

Напомним, что если второй блок с условием пустой, то это эквивалентно истинному условию во втором блоке. Поскольку такое «пустое» условие изменить уже нельзя, то формально мы получаем бесконечный цикл. Но его можно остановить. Для этого использована инструкция `break`. Она задействована вместе с условным оператором. В теле оператора цикла, после выполнения команд `s+=2*k-1` и `k++`, выполняется условный оператор. В нем проверяется условие `k>n`. Если условие истинно, то инструкцией `break` завершается выполнение оператора цикла.

> **НА ЗАМЕТКУ**
>
> Условие `k<=n`, которое мы размещали во втором блоке в операторе цикла `for`, было условием продолжения работы оператора цикла. Условие `k>n`, которое мы использовали в условном операторе, является условием завершения оператора цикла. То есть условие `k>n` является «противоположным» к условию `k<=n`.

Во всех рассмотренных примерах результат выполнения один и тот же (см. результат выполнения программы из листинга 3.10).

Инструкция безусловного перехода goto

Нормальные герои всегда идут в обход.

из к/ф «Айболит-66»

Инструкция *безусловного перехода* `goto` позволяет выполнить переход к определенному месту программного кода, помеченному с помощью *метки*. Метка — это обычный идентификатор, размещаемый в программном коде. Метку не нужно объявлять. Она просто указывается, и после нее следует двоеточие. Если в программном коде воспользоваться

инструкцией goto, указав после нее метку из программного кода, то выполнение такой команды приведет к тому, что следующей будет выполняться команда в строке, выделенной с помощью метки.

Таким образом, инструкция goto позволяет «прыгать» из одного места программного кода к другому, и это не может не вызывать некоторых опасений. Стоит признать, что они небезосновательны.Существует устоявшееся мнение, что наличие в программном коде команд с инструкцией goto — признак дурного тона в программировании. Другими словами, использование инструкции goto — это не тот путь, которым следует идти. Но прежде чем отказаться от использования какого-либо механизма, интересно хотя бы поверхностно оценить его возможности.

НА ЗАМЕТКУ

Причина такого недоброго отношения к инструкции goto связана с тем, что программные коды, написанные с использованием этой инструкции, обычно запутаны, сложны для анализа, да и быстрота их выполнения вызывает нарекания.

Как иллюстрацию к использованию инструкции goto мы рассмотрим программу, в которой традиционно вычисляется сумма нечетных чисел. Но на этот раз операторы цикла использовать не будем, а прибегнем к помощи условного оператора и, как несложно догадаться, инструкции goto. Интересующий нас программный код представлен в листинге 3.15.

Листинг 3.15. Использование инструкции goto

```
using System;
class GotoDemo{
   static void Main(){
      // Переменные: количество слагаемых, индексная
      // переменная и значение суммы:
      int n=10, k=1, s=0;
      // Отображение сообщения:
      Console.Write("Сумма 1 + 3 + 5 +…+ {0} = ",2*n-1);
      // Использование метки:
```

```
      mylabel:
      // Добавляем слагаемое в сумму:
      s+=2*k-1;
      // Изменение значения индексной переменной:
      k++;
      // Использование инструкции goto:
      if(k<=n) goto mylabel;  // Переход к помеченному коду
      // Отображение результата вычислений:
      Console.WriteLine(s);
   }
}
```

Результат выполнения этой программы такой же, как и в случаях, когда мы использовали операторы цикла.

Результат выполнения программы (из листинга 3.15)

```
Сумма 1 + 3 + 5 +…+ 19 = 100
```

Как же выполняется этот программный код? В принципе, все достаточно просто. В программе использованы, как и ранее, три целочисленные переменные `n`, `k` и `s`, назначение которых не изменилось — это количество слагаемых, индексная переменная и значение суммы чисел. Переменные получают начальные значения при объявлении. В программе использована метка `mylabel`. Это простой идентификатор, который заканчивается двоеточием. Само по себе наличие метки в программном коде никаких ощутимых последствий не имеет. То есть факт наличия метки на логику выполнения программного кода не влияет. На выполнение программного кода влияет наличие инструкции `goto`. Более конкретно, код выполняется следующим образом. После объявления и инициализации переменных `n`, `k` и `s` и отображения сообщения о вычислении суммы (команда `Console.Write("Сумма 1 + 3 + 5 +…+ {0} = ", 2*n-1)`) выполняются команды `s+=2*k-1` (добавление слагаемого в сумму) и `k++` (увеличение значения переменной на единицу). Вообще, поскольку данные команды не находятся в теле оператора цикла, то они должны были бы выполняться только один раз. Интрига появляется, когда дело доходит до выполнения условного оператора. В нем проверяется условие `k<=n`. Если условие истинно, то выполняется команда `goto mylabel`. Эта команда означает, что продолжать выполнение

программы следует с того места, которое выделено меткой `mylabel`. Несложно заметить, что сразу после метки `mylabel` находится команда `s+=2*k-1`. Поэтому в результате перехода из-за команды `goto mylabel` снова будет выполняться команда `s+=2*k-1`, а затем и команда `k++`. После этого на сцену опять выходит условный оператор, и при истинном условии `k<=n` выполнение программы будет продолжено с места, выделенного меткой `mylabel`. Поскольку каждый раз значение переменной `k` увеличивается на единицу, то в какой-то момент при проверке условия `k<=n` оно окажется ложным, и команда `goto mylabel` выполнена не будет. Тогда выполнится следующая команда после условного оператора, которой отображается вычисленное значение для суммы нечетных чисел. Таким образом, получается, что с помощью условного оператора и инструкции `goto` мы организовали некое подобие оператора цикла, хотя ни один из операторов цикла использован не был.

НА ЗАМЕТКУ

В плане алгоритма выполнения программы с инструкцией `goto` она аналогична программе для вычисления суммы нечетных чисел, реализованной с использованием оператора цикла `do-while` (см. листинг 3.9).

Перехват исключений

> Граждане, если вы увидите Бармалея, который собирается делать добрые дела, схватите его и поставьте в угол! Очень вас об этом прошу.
>
> *из к/ф «Айболит-66»*

Непосредственного отношения к управляющим инструкциям *обработка исключений* не имеет. Вообще, перехват исключений — тема большая и многогранная. Ей посвящена отдельная глава второй части книги. Здесь мы только кратко познакомимся с тем, как в языке C# на наиболее простом уровне может осуществляться перехват и обработка исключений. Причин для такого «преждевременного» знакомства две. Во-первых, те приемы, которые мы рассмотрим далее, полезны в прикладном смысле. С их помощью можно значительно повысить надежность и область применимости программных кодов. Во-вторых, некоторая аналогия между обработкой ошибок и управляющими инструкциями все же

есть — в определенном смысле это напоминает синтаксическую конструкцию, подобную условному оператору. Но если в условном операторе проверяется логическое выражение (условие), то при обработке исключений роль «условия» играет ошибка, которая может возникнуть, а может не возникнуть.

НА ЗАМЕТКУ

Впоследствии, когда мы познакомимся с искусственным генерированием исключений, мы увидим, что аналогии между управляющими инструкциями и обработкой исключений еще более глубокие.

Так что же такое исключение и что с ним можно делать? Дело в том, что при выполнении программы могут возникать ошибки. Речь не идет об ошибках, связанных с неправильно написанным программным кодом. Речь об ошибках, которые связаны с тем или иным режимом выполнения программы. Например, программа выдает запрос для ввода пользователем числа. Но ведь нет никакой гарантии, что пользователь введет именно число. В таком случае, когда программа попытается преобразовать введенное пользователем значение в число, может возникнуть ошибка. И возникнет она или нет — всецело зависит от пользователя, а не от программиста. Что может сделать программист? Программист может предусмотреть реакцию программы в случае, если пользователь вводит не число. Вот здесь и окажется полезной обработка исключительных ситуаций.

ПОДРОБНОСТИ

Если при выполнении программы происходит ошибка (говорят, что генерируется исключение), то в соответствии с типом ошибки создается специальный объект, который называется объектом ошибки, или исключением. Программу можно «научить» реагировать на ошибки. В таких случаях речь идет о перехвате и обработке исключений. То есть обработка исключений — это «комплекс мероприятий», выполняемых в связи с тем, что в программе в процессе выполнения возникла ошибка. Если в программе обработка исключений не выполняется, то возникновение ошибки приводит к завершению выполнения программы.

Итак, исходная постановка задачи формулируется следующим образом. Имеется некоторый программный код, при выполнении которого может

возникнуть ошибка. Необходимо так организовать программу, чтобы даже в случае возникновения ошибки она продолжала выполнение. Для решения этой задачи есть хороший рецепт.

НА ЗАМЕТКУ

Еще раз подчеркнем, что здесь мы только знакомимся с механизмом обработки исключений. Мы рассмотрим наиболее простой способ организовать такую обработку и не будем особо вдаваться в подробности. Обработке исключений посвящена отдельная глава второй части книги.

Идея состоит в том, чтобы использовать `try-catch` конструкцию. А именно, программный код, при выполнении которого может возникнуть ошибка (*контролируемый код*), помещается в `try`-блок. Делается это так: указывается ключевое слово `try`, а блок программного кода размещается в фигурных скобках после этого ключевого слова. После `try`-блока размещается `catch`-блок: указывается ключевое слово `catch` и после него в фигурных скобках — программный код, предназначенный для обработки исключений. Вся конструкция выглядит так, как показано ниже (жирным шрифтом выделены ключевые элементы шаблона):

```
try{
   // Контролируемый код
}
catch{
   // Код для обработки исключений
}
```

Работает `try-catch` конструкция так. Если при выполнении программного кода в `try`-блоке ошибка не возникает, то программный код в `catch`-блоке игнорируется и после завершения выполнения кода в `try`-блоке начинает выполняться команда после `try-catch` конструкции. Проще говоря, если ошибок нет, то наличие `catch`-блока никак не проявляется. Но ситуация резко меняется, если при выполнении кода в `try`-блоке возникает ошибка. Происходит следующее: если при выполнении какой-то команды возникает ошибка, то на этом выполнение команд в `try`-блоке прекращается и начинается выполнение команд в `catch`-блоке. После того как команды в `catch`-блоке будут выполнены, начинают выполняться команды после `try-catch` конструкции.

Самое важное во всей этой схеме: то, что, как бы ни развивались события, программа продолжает работу в штатном режиме. И это не может не радовать.

Как описанная схема используется на практике, проиллюстрировано в рассматриваемой далее программе. В этой программе сначала отображается диалоговое окно с сообщением о начале выполнения программы. Затем пользователю предлагается в окне с полем ввода указать действительное число. Если пользователь вводит число, то появляется соответствующее сообщение. Если пользователь не вводит ничего или вводит не число, появляется сообщение о том, что нужно было ввести число. Но в любом случае (ввел пользователь число или нет) перед завершением выполнения программы появляется еще одно сообщение. Теперь проанализируем программный код из листинга 3.16.

Листинг 3.16. Знакомство с перехватом исключений

```
using System;
using System.Windows.Forms;
using Microsoft.VisualBasic;
class TryCatchDemo{
   static void Main(){
      // Сообщение о начале выполнения программы:
      MessageBox.Show("Выполняется программа!","Начало");
      // Перехват и обработка исключений:
      try{// Контролируемый код
         // Попытка преобразовать текст в число:
         Double.Parse(
            Interaction.InputBox(
               // Текст над полем ввода:
               "Введите действительное число:",
               // Заголовок окна:
               "Число"
            )
         );
         // Отображение сообщения:
```

```
            MessageBox.Show("Да, это было число!","Число");
        }
        // Блок обработки исключений:
        catch{
            // Отображение сообщения:
            MessageBox.Show(
                // Текст сообщения:
                "Надо было ввести число!",
                // Заголовок окна:
                "Ошибка",
                // В окне одна кнопка OK:
                MessageBoxButtons.OK,
                // Используется пиктограмма ошибки:
                MessageBoxIcon.Error
            );
        } // Завершение блока обработки исключений
        // Сообщение о завершении выполнения программы:
        MessageBox.Show("Программа завершена!","Завершение");
    }
}
```

В программе командой `MessageBox.Show("Выполняется программа!","Начало")` отображается диалоговое окно с сообщением о том, что программа начала выполнение. Окно, которое появится на экране, показано на рис. 3.19.

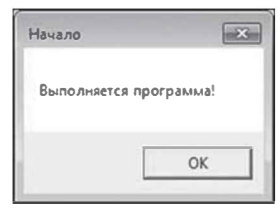


Рис. 3.19. Окно, которое отображается в начале выполнения программы

Затем выполняются команды в `try`-блоке (команды выполняются после того, как пользователь закрывает первое диалоговое окно с сообщением

о начале выполнения программы). Команд в `try`-блоке всего две. Первой командой отображается диалоговое окно с полем ввода, считывается введенное пользователем значение (в текстовом формате) и выполняется попытка преобразовать введенное значение в число с плавающей точкой (действительное число). Диалоговое окно с полем для ввода числа показано на рис. 3.20.

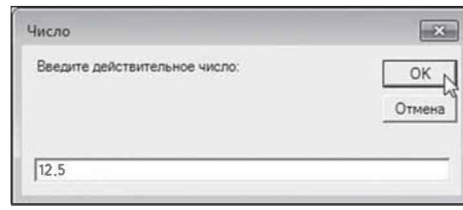


Рис. 3.20. Окно с полем для ввода действительного числа

НА ЗАМЕТКУ

В данном случае речь идет о вводе действительного числа. Не будет ошибкой, если пользователь введет целое число. Но также можно ввести и число в формате с плавающей точкой. Используемый в качестве «точки» символ — точка или запятая — определяется настройками операционной системы и среды разработки. На рис. 3.20 в качестве «точки» использована запятая.

Для отображения диалогового окна использован метод `InputBox()`. Результатом метод возвращает текстовую строку со значением, введенным пользователем. Команда вызова метода `InputBox()` указана аргументом статического метода `Parse()` структуры `Double` (структура из пространства имен `System`). Метод пытается выполнить преобразование текстового представления числа в число. Если попытка удачная, то результатом метода является соответствующее число. Если же пользователь не ввел число (или ввел не число), то возникает ошибка.

ПОДРОБНОСТИ

Мы знакомы со статическим методом `Parse()` из структуры `Int32`. Этот метод позволяет преобразовать текстовое представление целого числа в целое число. Метод с таким же названием, но уже из структуры `Double`, позволяет преобразовать текстовое представление для действительного числа в действительное число. Метод

возвращает результат — число, полученное в результате преобразования. То есть результат вызова метода можно записать в переменную. В рассматриваемом примере мы этого не делаем, поскольку нас само число не интересует. Интерес представляет только сам факт того, что пользователь ввел число.

Таким образом, если вызов метода `Parse()` из структуры `Double` прошел удачно, то пользователь ввел число. В этом случае в `try`-блоке выполняется следующая команда `MessageBox.Show("Да, это было число!","Число")`, которой отображается окно с сообщением о том, что было введено число. Окно показано на рис. 3.21.

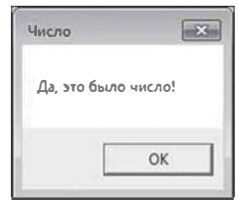


Рис. 3.21. Окно, которое отображается в случае, если пользователь ввел число

Если пользователь закрыл окно с полем ввода, щелкнув кнопку **Отмена** или системную пиктограмму, либо если он ввел не число, то при попытке получить числовое значение с помощью метода `Parse()` возникнет ошибка. В этом случае до выполнения команды `MessageBox.Show("Да, это было число!","Число")` дело не дойдет. Выполнение блока `try` будет прекращено, и начнут выполняться команды в блоке `catch`. А в этом блоке всего одна команда, которой отображается диалоговое окно с сообщением об ошибке. Аргументами методу `MessageBox.Show()` передаются:

- текст сообщения `"Надо было ввести число!"`;
- текст заголовка для окна `"Ошибка"`;
- константа `MessageBoxButtons.OK`, означающая, что в окне будет всего одна кнопка **ОК**;
- константа `MessageBoxIcon.Error`, означающая, что в окне отображается пиктограмма ошибки.

Как будет выглядеть отображаемое окно, показано на рис. 3.22.

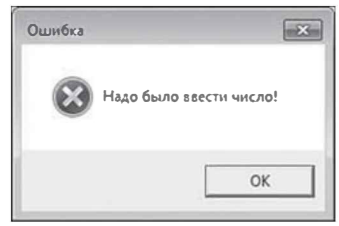


Рис. 3.22. Окно, которое отображается, если пользователь ввел не число или отказался от ввода числа

Но вне зависимости от того, произойдет или не произойдет ошибка, в конце, перед завершением выполнения программы, командой `MessageBox.Show("Программа завершена!","Завершение")` отображается диалоговое окно, представленное на рис. 3.23.

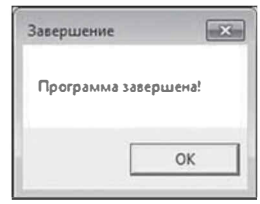


Рис. 3.23. Окно, которое отображается перед завершением выполнения программы

Мы видим, что даже в таком упрощенном варианте обработка исключений — достаточно мощный механизм, значительно повышающий гибкость и надежность программного кода. Как мы узнаем впоследствии, возможности этого механизма в действительности намного шире. Например, можно по-разному выполнять обработку исключений разного типа. Но об этом речь пойдет немного позже.

Резюме

> Пацак пацака не обманывает. Это некрасиво, родной.
>
> *из к/ф «Кин-дза-дза»*

- В языке C# существуют специальные управляющие инструкции, позволяющие создавать в программе точки ветвления и блоки повторяемых команд.
- Условный оператор `if` позволяет выполнять разные блоки команд в зависимости от истинности или ложности некоторого условия.

Проверяемое условие указывается в круглых скобках после ключевого слова `if`. Команды, выполняемые при истинном условии, указываются в блоке после `if`-инструкции. Команды, выполняемые при ложном условии, указываются в `else`-блоке. Существует упрощенная форма условного оператора без `else`-блока.

- Оператор выбора `switch` позволяет выполнять разные блоки команд в зависимости от значения некоторого выражения. Проверяемое выражение (целочисленное, символьное или текстовое) указывается в круглых скобках после ключевого слова `switch`. Затем указываются `case`-блоки с контрольными значениями. Выполняются команды в `case`-блоке, в котором контрольное значение совпадает со значением проверяемого выражения. В случае, если значение выражения не совпадает ни с одним из контрольных значений в `case`-блоках, выполняются команды в `default`-блоке. Этот блок не является обязательным. Каждый `case`-блок и `default`-блок заканчивается инструкцией `break`. В случае необходимости можно использовать пустые `case`-блоки.

- Оператор цикла `while` позволяет многократно выполнять блок определенных команд. После ключевого слова `while` в круглых скобках указывается условие, при истинности которого выполняются команды в теле оператора цикла. Каждый раз после выполнения этих команд проверяется условие, и, если оно истинно, команды выполняются снова.

- Оператор цикла `do-while` похож на оператор цикла `while`, но в операторе `do-while` сначала выполняются команды, а затем проверяется условие. Команды указываются после ключевого слова `do`. После блока команд следует ключевое слово `while` и, в круглых скобках, условие. Оператор цикла выполняется до тех пор, пока при очередной проверке условия оно не оказывается ложным.

- Описание оператора цикла `for` начинается с ключевого слова `for`. В круглых скобках указывается три блока инструкций. Блоки разделяются между собой точкой с запятой. Если блок содержит несколько инструкций, то они разделяются запятыми. Команды, формирующие тело оператора цикла, указываются в круглых скобках после `for`-инструкции. Выполнение оператора цикла начинается с выполнения команд в первом блоке. После этого проверяется условие во втором блоке. Если оно ложно, оператор цикла завершает работу. Если условие истинно, то выполняются команды в теле оператора

цикла и в третьем блоке. Затем снова проверяется условие. Если условие ложно, работа оператора цикла завершается. Если условие истинно, выполняются команды в теле оператора и в третьем блоке и снова проверяется условие. И так далее, пока при проверке условия оно не окажется ложным.

- Инструкция безусловного перехода `goto` позволяет перейти к выполнению программного кода в том месте, которое выделено меткой. Используя инструкцию `goto` и условный оператор, можно организовать циклическое выполнение программного кода (симулировать работу оператора цикла). Общая рекомендация состоит в том, чтобы избегать использования инструкции `goto`.
- Система обработки исключений позволяет предусмотреть специальный код, выполняемый при возникновении ошибки. С этой целью используется конструкция `try-catch`. Программный код, при выполнении которого может возникнуть ошибка, помещается в `try`-блок. Программный код, предназначенный для выполнения в случае возникновения ошибки, помещается в `catch`-блок. Если при выполнении кода в `try`-блоке ошибка не возникает, то `catch`-блок игнорируется. Если при выполнении кода в `try`-блоке возникает ошибка, то выполнение команд в блоке `try` прекращается и начинают выполняться команды в блоке `catch`.

Задания для самостоятельной работы

> Владимир Николаевич, у тебя дома жена, сып-двоечник, за кооперативпую квартиру не заплачено. А ты тут мозги пудришь. Плохо кончится, родной.
>
> *из к/ф «Кин-дза-дза»*

1. Напишите программу, в которой пользователь вводит число, а программа проверяет, делится ли это число на `3` и на `7`. Результаты проверки отображаются в сообщении в диалоговом окне. Используйте обработку исключений.

2. Напишите программу, в которой пользователь последовательно вводит два целых числа. Программа определяет, какое из чисел больше или они равны, и выводит сообщение в диалоговом окне. Используйте обработку исключений.

3. Напишите программу, в которой вычисляется сумма чисел, которые вводит пользователь. Программа выводит запрос на ввод числа, считывает введенное пользователем число, прибавляет его к сумме и снова выводит запрос на ввод числа. Процесс продолжается до тех пор, пока пользователь не введет нулевое значение. Используйте обработку исключений.

4. Напишите программу, в которой пользователь вводит целое число в диапазоне от `1` до `7`, а программа определяет по этому числу день недели. Если введенное пользователем число выходит за допустимый диапазон, выводится сообщение о том, что введено некорректное значение. Используйте оператор выбора `switch`. Предложите механизм обработки ошибки, связанной с вводом нечислового значения.

5. Напишите программу, в которой пользователю предлагается ввести название дня недели. По введенному названию программа определяет порядковый номер дня в неделе. Если пользователь вводит неправильное название дня, программа выводит сообщение о том, что такого дня нет. Предложите версию программы на основе вложенных условных операторов и на основе оператора выбора `switch`.

6. Напишите программу, в которой вычисляется сумма нечетных чисел. Для проверки результата воспользуйтесь тем, что $2 + 4 + 6 + ... + 2n = n(n + 1)$. Предложите версии программы, использующие разные операторы цикла.

7. Напишите программу для вычисления суммы квадратов натуральных чисел. Для проверки результата воспользуйтесь тем, что $1^2 + 2^2 + 3^2 + ... + n^2 = \frac{n(n+1)(2n+1)}{6}$. Предложите версии программы, использующие разные операторы цикла.

8. Напишите программу, которая выводит последовательность чисел Фибоначчи. Первые два числа в этой последовательности равны `1`, а каждое следующее число равно сумме двух предыдущих (получается последовательность `1`, `1`, `2`, `3`, `5`, `8`, `13`, `21`, `34`, `55`, `89` и так далее). Количество чисел в последовательности вводится пользователем. Предложите версии программы, использующие разные операторы цикла.

9. Напишите программу, в которой пользователем вводится два целых числа. Программа выводит все целые числа — начиная с наименьшего (из двух введенных чисел) и заканчивая наибольшим (из двух введенных чисел). Предложите разные версии программы (с использованием

разных операторов цикла), а также механизм обработки исключений для этой программы.

10. Напишите программу, в которой вычисляется сумма чисел, удовлетворяющих таким критериям: при делении числа на `5` в остатке получается `2`, или при делении на `3` в остатке получается `1`. Количество чисел в сумме вводится пользователем. Программа отображает числа, которые суммируются, и значение суммы. Используйте обработку исключений. Предложите версии программы, использующие разные операторы цикла.

Глава 4
МАССИВЫ

— Пойдем простым логическим ходом.
— Пойдем вместе.

из к/ф «Ирония судьбы, или С легким паром»

В этой главе мы обсудим исключительно важную конструкцию — речь пойдет о *массивах*. Среди тем, которые мы рассмотрим, будут такие:

- одномерные массивы — способы их объявления и использования;
- особенности работы с двумерными массивами;
- способы инициализации массивов;
- выполнение основных операций с массивами — в частности, речь будет идти о копировании и присваивании массивов;
- создание «рваных» массивов — то есть массивов со строками разной длины;
- особенности массива из объектных ссылок.

Также мы познакомимся со способами обработки аргументов командной строки. Еще в главе есть различные примеры использования массивов. Начнем же с азов — с создания одномерных массивов.

Одномерные массивы

Первый раз таких единоличников вижу.

из к/ф «Девчата»

Массив — это набор элементов одного типа, которые объединены общим именем. Переменные, входящие в массив, называются элементами массива. Поскольку одно и то же имя (имя массива) «применяется» сразу к нескольким переменным, то эти переменные нужно как-то

идентифицировать. Идентификация выполняется с помощью индекса или индексов. Индекс — это целое число. Количество индексов, необходимых для однозначной идентификации переменной в массиве, определяет *размерность* массива. Под *размером* массива обычно подразумевают общее количество элементов в массиве. Под размером массива для данной размерности имеют в виду количество значений, которые может принимать индекс, соответствующий данной размерности. Самые простые — *одномерные массивы*, в которых для идентификации элемента в массиве нужно указать всего один индекс. Знакомство с массивами начнем именно с одномерных массивов. То есть далее в этом разделе речь идет об одномерных массивах.

Мы уже выяснили, что массив — это набор переменных. Возникает естественный вопрос: как получить доступ к этому набору? Ответ состоит в том, что доступ к массиву получают с помощью специальной переменной, которая называется *переменной массива*. Как и обычную переменную, переменную массива перед использованием следует объявить. При объявлении переменной массива указывают идентификатор типа элементов массива, после него указываются пустые квадратные скобки и имя переменной массива. То есть шаблон объявления переменной массива такой:

```
тип[] переменная
```

Например, если мы хотим объявить переменную массива с названием `nums` и предполагается, что массив будет состоять из целочисленных значений типа `int`, то переменная массива объявляется командой `int[] nums`. Если бы мы объявляли переменную массива с названием `symbs` и массив предполагался состоящим из символьных значений, то команда объявления такой переменной выглядела бы как `char[] symbs`.

Но создание (объявление) переменной массива не означает создания массива. Другими словами, даже если мы объявили переменную массива, сам массив от этого не появляется. Его еще нет. Массив нужно создать.

Для создания массива используется оператор `new`. После оператора `new` указывается идентификатор типа, соответствующий типу элементов создаваемого массива, а в квадратных скобках после идентификатора типа указывается размер массива. Шаблон для команды создания массива имеет такой вид:

```
new тип[размер]
```

Например, если мы хотим создать одномерный целочисленный массив из 12 элементов, то команда создания такого массива выглядит как `new int[12]`. Чтобы создать массив из 10 элементов символьного типа, используем команду `new char[10]`. Остается открытым вопрос о том, как переменная массива связана с собственно массивом. Ответ очень простой: переменная массива ссылается на массив. Можно сказать и иначе: значением переменной массива может быть адрес массива в памяти.

НА ЗАМЕТКУ

Что понимать под «адресом массива» и как этот адрес используется, более детально обсуждается в главе (во второй части книги), посвященной работе с указателями. Пока же нам достаточно самых общих представлений о том, что такое «адрес массива».

Также нужно учесть, что команда создания массива не только создает массив, но еще и имеет результат. Это адрес созданного массива. А адрес массива, как мы уже знаем, может быть значением переменной массива. Главное, чтобы тип переменной массива соответствовал типу элементов в массиве. Если обобщить все вышеизложенное, то общая схема создания массива реализуется следующими командами:

```
тип[] переменная;
переменная=new тип[размер];
```

То есть задача по созданию массива состоит из двух этапов:

- объявление переменной массива;
- создание массива и присваивание ссылки на массив переменной массива.

Эти два этапа можно объединить и воспользоваться такой командой:

```
тип[] переменная=new тип[размер];
```

Схема реализации одномерного массива проиллюстрирована на рис. 4.1.

Например, командой `int[] nums=new int[12]` создается целочисленный массив из 12 элементов, объявляется переменная массива `nums` и ссылка на созданный массив записывается в эту переменную. Командой `char[] symbs=new char[10]` создается массив из 10 символьных элементов, объявляется переменная массива `symbs` и ссылка на массив записывается в переменную.

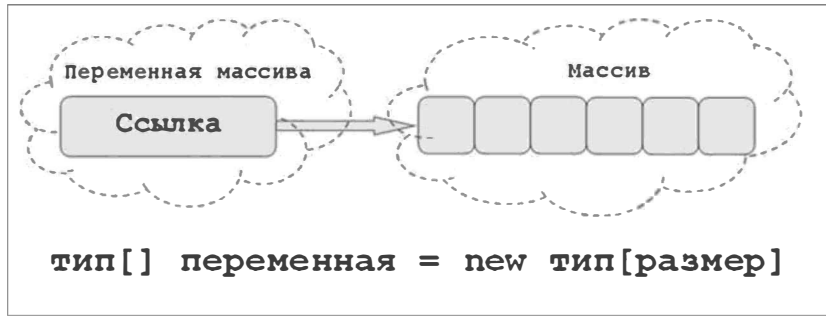


Рис. 4.1. Схема реализации одномерного массива

После того как массив создан, к его элементам можно обращаться для присваивания значений и считывания значений. Делается это достаточно просто: после имени массива (имя массива отождествляется с переменной массива, которая ссылается на этот массив) в квадратных скобках указывается индекс элемента. Важно помнить, что индексация элементов всегда начинается с нуля. Поэтому у первого элемента индекс `0`, у второго элемента индекс `1` и так далее. Индекс последнего элемента в массиве на единицу меньше размера массива. А размер массива можно определить с помощью свойства `Length`. Например, если мы имеем дело с одномерным массивом `nums`, то значением выражения `nums.Length` является количество элементов в массиве `nums`. Ссылка на первый элемент в массиве `nums` выглядит как `nums[0]`, ссылка на второй элемент — это `nums[1]`, а ссылка на последний элемента массива `nums` дается выражением `nums[nums.Length-1]`.

Как пример создания и использования массива рассмотрим программу в листинге 4.1. Там создается массив целых чисел, а затем этот массив заполняется числами, которые при делении на `3` в остатке дают `1` (то есть речь о числах `1`, `4`, `7`, `10`, `13`, `16` и так далее).

Листинг 4.1. Создание одномерного числового массива

```
using System;
class IntArrayDemo{
   static void Main(){
      // Создание массива из 12 чисел:
      int[] nums=new int[12];
      // Перебор элементов массива:
      for(int k=0; k<nums.Length; k++){
```

```
            // Присваивание значения элементу массива:
            nums[k]=3*k+1;
            // Отображение значения элемента массива:
            Console.Write("| "+nums[k]+" ");
        }
        Console.WriteLine("|");
    }
}
```

Результат выполнения программы такой, как показано ниже.

Результат выполнения программы (из листинга 4.1)

```
| 1 | 4 | 7 | 10 | 13 | 16 | 19 | 22 | 25 | 28 | 31 | 34 |
```

Программа очень проста. В теле главного метода командой `int[] nums=new int[12]` создается массив, состоящий из `12` целочисленных элементов. Ссылка на массив записывается в переменную массива `nums`. После создания массива его элементы существуют, но им необходимо присвоить значения. Для этого мы запускаем оператор цикла, в котором индексная переменная `k` принимает начальное значение `0`. Это значение индекса первого (начального) элемента в массиве.

НА ЗАМЕТКУ

Индексная переменная `k` объявлена в первом блоке в операторе цикла `for`. Поэтому переменная `k` доступна только в операторе цикла, но не за его пределами. Напомним общее правило, определяющее область доступности переменной: оно гласит, что переменная доступна в пределах того блока, в котором она объявлена. Для переменной `k` таким «блоком» является оператор цикла.

За каждую итерацию переменная `k` увеличивает свое значение на единицу (благодаря команде `k++` в третьем блоке оператора цикла `for`). Оператор цикла продолжает работу, пока истинно условие `k<nums.Length`. Оно означает, что значение индексной переменной `k` не должно превышать значение выражения `nums.Length`. Значение выражения `nums.Length` — это размер массива. Учитывая, что переменная `k` принимает только целочисленные значения, легко догадаться, что последнее значение переменной `k`, при котором еще будут выполняться команды

в теле оператора цикла, равно `nums.Length-1`. Это значение на единицу меньше размера массива и совпадает со значением индекса последнего элемента в массиве `nums` (напомним, что индекс последнего элемента в одномерном массиве всегда на единицу меньше размера этого массива). Таким образом, переменная `k` последовательно пробегает значения всех элементов массива `nums`. За каждый цикл в теле оператора цикла выполняются такие команды. Сначала командой `nums[k]=3*k+1` элементу с индексом `k` присваивается значение `3*k+1`. Здесь мы учли, что для чисел, которые при делении на `3` в остатке дают `1`, может быть использована формула $3k + 1$, где целое число k может принимать значения 0, 1, 2 и так далее. Легко проверить, что, используя эту формулу, последовательно получаем числа `1`, `4`, `7`, `10` и так далее.

НА ЗАМЕТКУ

Числа, которые при делении на число A дают в остатке число B, рассчитываются по формуле $Ak + B$, где параметр k может принимать значения 0, 1, 2 и так далее.

После того как значение очередному элементу массива присвоено, элемент (его значение) отображается в консольном окне. Для этого использована команда `Console.Write("| "+nums[k]+" ")` в теле оператора цикла. В консольное окно выводится вертикальная черта, пробел, значение элемента и еще один пробел. Причем, поскольку использован метод `Write()`, все сообщения выводятся в одну строку. В результате в консольном окне в одной строке появится последовательность значений элементов массива. Значения разделяются пробелами и вертикальными разделителями. Но после последнего элемента (и пробела) вертикальной черты не будет. Она добавляется командой `Console.WriteLine("|")` уже после завершения оператора цикла.

Еще один небольшой пример касается создания символьного массива (массива с элементами типа `char`). Размер массива определяется значением переменной. После создания массив заполняется случайными символами. Содержимое массива отображается в прямом и обратном порядке. Код программы представлен в листинге 4.2.

Листинг 4.2. Символьный массив

```
using System;
class CharArrayDemo{
```

```
   static void Main(){
      // Объект для генерирования случайных чисел:
      Random rnd=new Random();
      // Размер массива и индексная переменная:
      int size=10, k;
      // Создание символьного массива:
      char[] symbs=new char[size];
      // Отображение сообщения:
      Console.WriteLine("Массив случайных символов:");
      // Заполнение массива случайными символами:
      for(k=0; k<symbs.Length; k++){
         // Значение элемента массива:
         symbs[k]=(char)('A'+rnd.Next(26));
         // Отображение значения элемента массива:
         Console.Write("| "+symbs[k]+" ");
      }
      Console.WriteLine("|");
      // Отображение сообщения:
      Console.WriteLine("Массив в обратном порядке:");
      for(k=symbs.Length-1; k>=0; k--){
         // Отображение значения для элемента массива:
         Console.Write("| "+symbs[k]+" ");
      }
      Console.WriteLine("|");
   }
}
```

НА ЗАМЕТКУ

Вообще, сгенерировать случайное число — задача очень нетривиальная. Откровенно говоря, случайных чисел не существует вовсе. Есть иллюзия того, что числа случайные. В реальности такие числа генерируются в соответствии с определенным алгоритмом. И они конечно же не случайны. Поэтому более корректный термин — псевдослучайные числа.

Как отмечалось выше, в программе массив заполняется случайными символами. Для генерирования случайных символов нам понадобятся случайные целые числа.

Генерируются случайные числа с помощью специального *объекта* `rnd`, который создается на основе класса `Random` (из пространства имен `System`) командой `Random rnd=new Random()`. В результате выполнения этой команды мы получаем объект `rnd`, у которого есть метод `Next()`, возвращающий значением случайное целое число.

ПОДРОБНОСТИ

Классы и объекты, которые создаются на основе классов, обсуждаются позже. Сейчас отметим лишь основные, наиболее важные моменты, связанные с созданием объектов. Объект создается на основе класса. Класс играет роль шаблона, определяющего, какие поля, свойства и методы будет иметь объект. Фактически класс представляет собой специфический «тип данных» (с поправкой на то, что там не только данные). Объекты реализуются подобно массивам: есть собственно объект, а есть переменная, которая на этот объект ссылается. Последняя называется объектной переменной. Например, мы хотим создать объект на основе класса `Random`. Сам объект создается командой `new Random()`. У выражения `new Random()` есть значение — это ссылка на созданный объект. Ссылка записывается в объектную переменную `rnd`. Объектная переменная объявляется командой вида `Random rnd`, в которой в качестве типа переменной указывается название класса `Random`. Если две команды (объявления объектной переменной и создания объекта) объединить в одну, то получим выражение `Random rnd=new Random()`. Как переменная массива обычно отождествляется с массивом, так и объектная переменная, как правило, отождествляется с объектом.

Символьный массив создается командой `char[] symbs=new char[size]`. Размер массива определяется значением переменной `size`. Поскольку предварительно при объявлении переменной `size` ей было присвоено значение `10`, то создается массив из `10` элементов.

НА ЗАМЕТКУ

Размер массива определяется значением переменной `size` на момент создания массива. Если бы впоследствии значение переменной `size` изменилось, размер массива остался бы неизменным.

Для заполнения массива случайными символами мы используем оператор цикла. Индексная переменная `k` в операторе цикла принимает целочисленные значения от `0` и строго меньше `symbs.Length` (размер массива), увеличиваясь за каждый цикл на единицу. Значение элементу массива `symbs` с индексом `k` присваивается командой `symbs[k]=(char)('A'+rnd.Next(26))`, в которой генерируется случайное число. Значением выражения `rnd.Next(26)` является случайное целое число в диапазоне от `0` до `25` включительно.

ПОДРОБНОСТИ

Метод `Next()` вызывается из объекта класса `Random` (в нашем случае таким объектом является `rnd`). Есть несколько способов вызова метода `Next()`. Если метод вызывается с одним целочисленным аргументом, то результатом является неотрицательное случайное целое число, не превышающее то, которое указано аргументом. Например, если использована команда `rnd.Next(26)`, то результатом будет неотрицательное целое число, не большее `26` — это число в диапазоне значений от `0` до `25` включительно.

Следовательно, к символу `'A'` прибавляется целое число (речь о выражении `'A'+rnd.Next(26)`). Результат выражения вычисляется так: к коду символа `'A'` (значение `65`, но в данном случае это не важно) прибавляется целочисленное слагаемое. В итоге получаем целое число. Перед всем выражением указана инструкция приведения к символьному типу `(char)`. Поэтому полученное числовое значение интерпретируется как код символа в кодовой таблице. Поскольку в английском алфавите `26` букв, а в кодовой таблице буквы расположены в соответствии с алфавитом, то результатом выражения `(char)('A'+rnd.Next(26))` является одна из больших (прописных) букв. Поскольку к коду символа `'A'` прибавляется случайное число, то и буква будет случайной. Диапазон возможных значений — от `'A'` до `'Z'` включительно.

После того как значение элементу массива присвоено, командой `Console.Write("| "+symbs[k]+" ")` оно отображается в консольном окне. Как и в предыдущем примере, мы используем вертикальные разделители и пробелы при отображении значений элементов массива. Последний разделитель отображается командой `Console.WriteLine("|")` после завершения выполнения оператора цикла.

Для отображения содержимого массива в обратном порядке мы используем еще один оператор цикла. В нем индексная переменная `k`

(мы используем ту же переменную, что и в первом операторе цикла) за каждую итерацию уменьшает значение на единицу, начиная с начального значения `symbs.Length-1` (значение индекса последнего элемента в массиве `symbs`) до значения `0` включительно (использовано условие `k>=0`). За каждый цикл командой `Console.Write("| "+symbs[k]+" ")` отображается значение элемента с индексом `k`. Поскольку элементы в массиве перебираются в обратном порядке, то в консольном окне массив тоже отображается в обратном порядке (начиная с последнего элемента и заканчивая первым). При этом сам массив не меняется.

ПОДРОБНОСТИ

Индексная переменная `k` объявлена вместе с переменной `size` — то есть она объявлена не в операторе цикла. Область доступности переменной `k` — главный метод программы. Поэтому мы можем использовать переменную в обоих операторах цикла. Если бы мы объявили переменную `k` в первом операторе цикла, то во втором ее использовать уже не смогли бы. Но мы могли бы объявить во втором операторе цикла другую индексную переменную или даже переменную с таким же названием `k`. Даже если так, то это были бы разные переменные. Область доступности каждой из них ограничивалась бы «своим» оператором цикла.

Результат выполнения программы может быть следующим.

Результат выполнения программы (из листинга 4.2)

```
Массив случайных символов:
| F | U | Q | P | P | A | C | P | A | H |
Массив в обратном порядке:
| H | A | P | C | A | P | P | Q | U | F |
```

Поскольку в программе использован генератор случайных чисел, то от запуска к запуску значения у элементов массива могут (и будут) меняться. Неизменным остается тот факт, что во втором случае значения элементов отображаются в обратном порядке по сравнению с тем, как они отображались вначале.

Инициализация массива

Не мешайте работать, инвентаризуемый.

из к/ф «Гостья из будущего»

В рассмотренных примерах мы сначала создавали массив, а затем его заполняли. При заполнении массивов у нас был определенный алгоритм, в соответствии с которым элементам присваивались значения. Например, массив можно заполнять нечетными числами, или случайными числами, или числами Фибоначчи. Но бывают ситуации, когда такой закономерности попросту нет. Элементам массива нужно присвоить значения, и эти значения нельзя (или нет смысла) записывать в виде формулы. Проще говоря, очень часто принципиально невозможно заполнить массив с использованием оператора цикла. Нужно каждому элементу присваивать значение «в индивидуальном порядке». Это очень неудобно, особенно если массив большой. В таких случаях удобно выполнять явную *инициализацию* массива. Делается все достаточно просто: при объявлении переменной массива ей присваивается список со значениями, которые формируют массив. Значения в списке заключаются в фигурные скобки. Внутри скобок значения разделяются запятыми. Например, если мы воспользуемся командой `int[] nums={1,3,5}`, то в результате будет создан целочисленный массив из трех элементов. Значение первого элемента равно `1`, значение второго элемента равно `3`, значение третьего элемента равно `5`. Ссылка на созданный массив записывается в переменную `nums`.

ПОДРОБНОСТИ

Кроме команды `int[] nums={1,3,5}`, для создания и инициализации массива можно было бы воспользоваться командами `int[] nums=new int[3]{1,3,5}` или `int[] nums=new int[]{1,3,5}`. Разница между последними двумя командами в том, что в одной размер массива указан явно, а в другой — нет. Если размер массива не указан, то он автоматически определяется по количеству элементов в списке инициализации. Если размер массива указан, то он должен соответствовать количеству элементов в списке инициализации. В противном случае возникает ошибка.

Небольшой пример с инициализацией различных массивов приведен в программе в листинге 4.3.

Листинг 4.3. Инициализация одномерного массива

```
using System;
class InitArrayDemo{
   static void Main(){
      // Создание и инициализация массивов:
      int[] nums={1,3,5,7,6,5,4};
      char[] symbs=new char[]{'A','Z','B','Y'};
      string[] txts=new string[3]{"один","два","три"};
      // Отображение содержимого массивов:
      Console.WriteLine("Массив nums:");
      for(int k=0; k<nums.Length; k++){
         Console.Write(nums[k]+" ");
      }
      Console.WriteLine("\nМассив symbs:");
      for(int k=0; k<symbs.Length; k++){
         Console.Write(symbs[k]+" ");
      }
      Console.WriteLine("\nМассив txts:");
      for(int k=0; k<txts.Length; k++){
         Console.Write(txts[k]+" ");
      }
      Console.WriteLine();
   }
}
```

Результат выполнения программы такой, как показано ниже.

Результат выполнения программы (из листинга 4.3)

```
Массив nums:
1 3 5 7 6 5 4
Массив symbs:
A Z B Y
Массив txts:
один два три
```

В данном случае программный код очень прост. В нем последовательно создается и инициализируется три массива: целочисленный `nums`, символьный `symbs` и текстовый `txts`. Для каждого из этих массивов предусмотрен список со значениями, которыми инициализируются элементы массива.

НА ЗАМЕТКУ

Мы используем три оператора цикла, в каждом из которых объявляется индексная переменная с названием `k`. Это разные переменные, хотя формально они имеют одинаковые названия. Каждая такая переменная доступна в пределах оператора цикла, в котором она объявлена. Более того, такая переменная существует, пока выполняется оператор цикла: при запуске оператора цикла на выполнение под переменную выделяется место, а когда оператор цикла завершает работу — переменная удаляется (в памяти освобождается место, выделенное под эту переменную).

Операции с массивами

> Ведь это же настоящая тайна! Ты нотом никогда себе не простишь!
>
> *из к/ф «Гостья из будущего»*

Создание массива, заполнение массива и отображение значений его элементов — это далеко не все операции, которые можно выполнять с массивами. Область применения массивов намного шире. В этом разделе мы рассмотрим примеры, иллюстрирующие методы работы с массивами. Для начала проанализируем программный код в листинге 4.4.

Листинг 4.4. Копирование и присваивание массивов

```
using System;
class CopyArrayDemo{
   static void Main(){
      // Целочисленный массив:
      int[] A={1,3,5,7,9};
      // Переменные массива:
```

```
      int[] B, C;
      // Присваивание массивов:
      B=A;
      // Создание нового массива:
      C=new int[A.Length];
      // Поэлементное копирование массива:
      for(int k=0; k<A.Length; k++){
         C[k]=A[k];
      }
      // Изменение значения первого элемента в массиве A:
      A[0]=0;
      // Изменение значения последнего элемента
      // в массиве B:
      B[B.Length-1]=0;
      // Сообщение в консольном окне:
      Console.WriteLine("A:\tB:\tC:");
      // Отображение содержимого массивов:
      for(int k=0; k<A.Length; k++){
         // Отображение значений элементов массивов:
         Console.WriteLine("{0}\t{1}\t{2}", A[k], B[k], C[k]);
      }
   }
}
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 4.4)

```
A:      B:      C:
0       0       1
3       3       3
5       5       5
7       7       7
0       0       9
```

В программе командой `int[] A={1,3,5,7,9}` создается и инициализируется целочисленный массив `A` из `5` элементов (значения — нечетные числа от `1` до `9` включительно). А вот командой `int[] B, C` объявляются две переменные массива (но сами массивы не создаются!). Переменные `B` и `C` в силу того, как они объявлены (идентификатором типа указано выражение `int[]`), могут ссылаться на одномерные целочисленные массивы. Значение переменной `B` присваивается командой `B=A`. Но копирования массива `A` в данном случае не происходит. Чтобы понять, каковы последствия команды `B=A`, следует учесть, что в действительности значением переменной массива является ссылка или адрес массива. Поэтому значение переменной `A` — это адрес массива, на который ссылается переменная. Этот адрес копируется в переменную `B`. Как следствие, переменная `B` будет ссылаться на тот же массив, на который ссылается переменная `A`. Это важный момент: при выполнении команды `B=A` новый массив не создается. И переменная `A`, и переменная `B` будут ссылаться на физически один и тот же массив.

Иначе мы поступаем с переменной `C`. Командой `C=new int[A.Length]` создается новый пустой массив, и ссылка на этот массив записывается в переменную `C`. Размер созданного массива определяется выражением `A.Length`, и поэтому он такой же, как у массива `A`. Затем с помощью оператора цикла выполняется поэлементное копирование (команда `C[k]=A[k]` в теле оператора цикла) содержимого массива `A` в массив `C`. В результате мы получаем, что переменные `A` и `C` ссылаются на одинаковые массивы, но это все же два отдельных массива (хотя и с совпадающими значениями элементов).

При выполнении команды `A[0]=0` в массиве, на который ссылается переменная `A` (и переменная `B`), значение первого элемента меняется с `1` на `0`. При выполнении команды `B[B.Length-1]=0` значение последнего элемента в массиве, на который ссылается переменная `B` (и переменная `A`), меняется с `9` на `0`. Но в массиве `C` все элементы остались с теми же значениями, которые у них были в самом начале после поэлементного копирования. В частности, первый элемент остался со значением `1`, а последний элемент остался со значением `9`. Это подтверждает результат выполнения программы.

ПОДРОБНОСТИ

В команде `Console.WriteLine("A:\tB:\tC:")` в текстовом литерале использована инструкция табуляции `\t`. Инструкция обрабатывается так. При отображении текста в консоли каждый

символ отображается в определенной позиции. Они идут подряд одна за другой. Среди них есть особые «реперные» — как правило, через каждые восемь позиций. То есть «реперными» является 1-я, 9-я, 17-я, 25-я позиция и так далее. Если при отображении текста встречается инструкция `\t`, то следующий символ будет отображаться в ближайшей реперной позиции (вправо от позиции последнего отображенного символа). На практике применение инструкции табуляции `\t` позволяет выполнять выравнивание по столбикам.

В теле оператора цикла, в котором отображаются значения элементов массивов `A`, `B` и `C` (точнее, массивов, на которые ссылаются указанные переменные — поскольку массивов всего два), использована команда `Console.WriteLine("{0}\t{1}\t{2}", A[k], B[k], C[k])`. При отображении текстовой строки, указанной первым аргументом метода `WriteLine()`, вместо инструкции `{0}` подставляется значение элемента `A[k]` (при заданном значении индекса `k`), вместо инструкции `{1}` подставляется значение элемента `B[k]`, а вместо инструкции `{2}` подставляется значение элемента `C[k]`. После отображения первого и второго значения выполняется табуляция. В итоге значения элементов массива, на который ссылается переменная `A`, отображаются в столбике под надписью `"A:"`. Под надписью `"B:"` в столбик отображаются значения элементов массива, на который ссылается переменная `B`. А под надписью `"C:"` в столбик отображаются значения элементов массива, на который ссылается переменная `C`.

Таким образом, при копировании значений переменных массива ссылка из одной переменной копируется в другую. В результате обе переменные будут ссылаться на один и тот же массив.

Как пример обработки содержимого массива рассмотрим задачу о поиске наибольшего элемента в числовом массиве. Соответствующая программа представлена в листинге 4.5.

Листинг 4.5. Поиск наибольшего значения в массиве

```
using System;
class MaxElementDemo{
   static void Main(){
      // Переменные для записи значения элемента и индекса:
      int value, index;
      // Размер массива:
```

```
      int size=15;
      // Объект для генерирования случайных чисел:
      Random rnd=new Random();
      // Создание массива:
      int[] nums=new int[size];
      // Заполнение и отображение массива:
      for(int k=0; k<nums.Length; k++){
         // Значение элемента массива:
         nums[k]=rnd.Next(1,101);
         // Отображение значения элемента:
         Console.Write(nums[k]+" ");
      }
      Console.WriteLine();
      // Поиск наибольшего элемента:
      index=0;           // Начальное значение для индекса
      value=nums[index]; // Значение элемента с индексом
      // Перебор элементов:
      for(int k=1; k<nums.Length; k++){
         // Если значение проверяемого элемента больше
         // текущего наибольшего значения:
         if(nums[k]>value){
            value=nums[k]; // Новое наибольшее значение
            index=k;       // Новое значение для индекса
         }
      }
      // Отображение результата:
      Console.WriteLine("Наибольшее значение: "+value);
      Console.WriteLine("Индекс элемента: "+index);
   }
}
```

В программе создается целочисленный массив, заполняется случайными числами (в диапазоне значений от 1 до 100 включительно), а после

этого выполняется поиск элемента с наибольшим значением. Программой в консольное окно выводится сообщение о том, какое максимальное значение и какой индекс у элемента с максимальным значением.

НА ЗАМЕТКУ

Если в массиве несколько элементов с максимальным значением, то выводится индекс первого из этих элементов.

Результат выполнения программы может быть таким (с поправкой на использование генератора случайных чисел).

Результат выполнения программы (из листинга 4.5)

```
43 4 40 30 41 18 77 70 91 15 40 86 9 55 38
Наибольшее значение: 91
Индекс элемента: 8
```

Для запоминания значения элемента и его индекса используются целочисленные переменные `value` и `index`. Размер массива, который создается и заполняется случайными числами, определяется значением целочисленной переменной `size` (значение переменной равно `15` — то есть создается массив из `15` элементов). Для генерирования случайных чисел с помощью команды `Random rnd=new Random()` создаем объект `rnd` класса `Random`.

НА ЗАМЕТКУ

Объект создается инструкцией `new Random()`, а ссылка на этот объект записывается в объектную переменную `rnd`. После этого переменную `rnd` можно отождествлять с данным объектом. У объекта есть метод `Next()`, возвращающий результатом случайное целое число. Диапазон возможных значений определяется аргументом (или аргументами) метода.

Целочисленный массив создается командой `int[] nums=new int[size]`. После этого для заполнения массива запускается оператор цикла, в котором локальная индексная переменная `k` последовательно перебирает значения индексов элементов массива. За каждый цикл выполняется команда `nums[k]=rnd.Next(1,101)`, которой элементу массива `nums` с индексом `k` значением присваивается случайное целое число в диапазоне от `1` до `100` включительно.

ПОДРОБНОСТИ

Если при вызове методу `Next()` передается неотрицательное целое число, то результатом является случайное целое число в диапазоне от `0` и строго меньше числа, переданного аргументом методу. Если аргументом методу `Next()` переданы два неотрицательных целых числа, то результатом является случайное число, большее или равное первому аргументу и строго меньшее второго аргумента. Например, если мы вызываем метод `Next()` с аргументами `1` и `101` (команда `rnd.Next(1,101)`), то в результате будет сгенерировано число в диапазоне возможных значений от `1` до `100` включительно.

После присваивания значения элементу массива это значение отображается в консольном окне командой `Console.Write(nums[k]+" ")`.

После того как массив создан и заполнен, начинается поиск элемента с наибольшим значением. Алгоритм используется следующий. Мы рассматриваем начальный элемент в массиве как такой, что имеет максимальное значение. Это первый «претендент» на звание элемента с наибольшим значением. Затем последовательно перебираем все прочие элементы, и, как только находим элемент с большим значением, он занимает место «претендента». В контексте этого алгоритма командой `index=0` переменной `index` значением присваивается начальное нулевое значение — это индекс первого элемента в массиве. Переменной `value` значение присваивается командой `value=nums[index]`. Это значение первого элемента в массиве. После присваивания начальных значений переменным `value` и `index` запускается оператор цикла, в котором перебираются все элементы массива, кроме начального. Поэтому в этом операторе цикла локальная индексная переменная `k` принимает значения, начиная с `1` (а не с `0`, как было ранее). В теле оператора цикла размещен условный оператор. В условном операторе проверяется условие `nums[k]>value`. Состоит оно в том, что проверяемый элемент массива `nums[k]` имеет значение большее, чем текущее значение в переменной `value`. Если условие истинно, то командой `value=nums[k]` переменной `value` присваивается новое значение (значение проверяемого элемента массива). Также командой `index=k` присваивается новое значение переменной `index` (это индекс проверяемого элемента).

Таким образом, после завершения выполнения оператора цикла в переменную `value` записано значение наибольшего элемента в массиве,

а в переменную `index` записано значение индекса этого элемента (если таких элементов несколько, то индекс первого из них). Командами `Console.WriteLine("Наибольшее значение: "+value)` и `Console.WriteLine("Индекс элемента: "+index)` вычисленные значения отображаются в консольном окне.

> **НА ЗАМЕТКУ**
>
> Если в условии в условном операторе оператор *«больше»* > заменить на *«больше или равно»* >=, то программа будет работать так же, за исключением одного момента. Если в массиве несколько элементов с наибольшим значением, то будет вычисляться индекс последнего из этих элементов.

Следующий пример имеет отношение к сортировке массивов. Для этого используем *метод пузырька.* Метод пузырька может применяться к массиву из элементов, для которых имеют смысл операции сравнения. Сортировка массива выполняется следующим образом. Последовательно перебираются все элементы массива. Сравниваются пары соседних элементов. Если значение элемента слева больше значения элемента справа, то эти элементы обмениваются значениями. В противном случае ничего не происходит. После одного полного перебора элементов самое большое значение гарантированно будет у последнего элемента. Чтобы второе по величине значение было у предпоследнего элемента, необходимо еще раз перебрать элементы массива (и выполнить для них попарное сравнение). При этом последний элемент уже можно не проверять. Чтобы переместить на «правильную» позицию третье по величине значение, еще раз перебираются элементы массива (теперь можно не проверять два последних элемента). Процесс продолжается до тех пор, пока значения элементов не будут отсортированы в возрастающем порядке. Итак, один полный перебор элементов гарантирует перестановку в «правильную» позицию одного значения. Количество полных переборов на единицу меньше количества элементов в массиве, поскольку, когда второй элемент имеет «правильное» значение, то автоматически у первого элемента оно тоже «правильное». При первом переборе проверяются все элементы массива. За каждый следующий перебор проверяется на один элемент меньше, чем это было для предыдущего раза.

Теперь рассмотрим программный код в листинге 4.6. Там методом пузырька сортируется символьный массив.

Листинг 4.6. Сортировка массива методом пузырька

```
using System;
class SortArrayDemo{
   static void Main(){
      // Символьная переменная:
      char s;
      // Исходный символьный массив:
      char[] symbs={'Q','Ы','a','B','R','A','r','q','b'};
      // Отображение содержимого массива:
      Console.WriteLine("Массив до сортировки:");
      for(int k=0; k<symbs.Length; k++){
         Console.Write(symbs[k]+" ");
      }
      Console.WriteLine();
      // Сортировка элементов в массиве:
      for(int i=1; i<symbs.Length; i++){
         // Перебор элементов:
         for(int j=0; j<symbs.Length-i; j++){
            // Если значение элемента слева больше
            // значения элемента справа:
            if(symbs[j]>symbs[j+1]){
               s=symbs[j+1];
               symbs[j+1]=symbs[j];
               symbs[j]=s;
            }
         }
      }
      // Отображение содержимого массива:
      Console.WriteLine("Массив после сортировки:");
      for(int k=0; k<symbs.Length; k++){
         Console.Write(symbs[k]+" ");
      }
```

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

Как выглядит результат выполнения программы, показано ниже.

Результат выполнения программы (из листинга 4.6)

```
Массив до сортировки:
Q Ы a B R A r q b
Массив после сортировки:
A B Q R a b q r Ы
```

Массив, предназначенный для сортировки, создается в программе командой `char[] symbs={'Q','Ы','a','B','R','A','r','q','b'}`. Это символьный массив. Он инициализирован списком, содержащим строчные и прописные английские буквы и одну кириллическую (символ `'Ы'`). Массив перед сортировкой отображается в консольном окне. Для этой цели использован оператор цикла. После этого начинается сортировка элементов массива. Для сортировки использованы вложенные операторы цикла. Внешний оператор цикла с индексной переменной `i` «подсчитывает» общие переборы элементов в массиве. Значение индексной переменной `i` последовательно меняется с `1` до `symbs.Length-1` включительно (то есть общее количество переборов на единицу меньше количества элементов в массиве). При фиксированном значении индексной переменной `i` во внутреннем операторе цикла индексная переменная `j` пробегает значения от `0` до `symbs.Length-i-1` включительно. Эта индексная переменная «перебирает» индексы элементов в массиве. За первый цикл (при значении переменной `i` равном `1`) переменная `j` пробегает значения от `0` до `symbs.Length-2`. Здесь нужно учесть, что в теле оператора цикла обрабатываются два элемента — с индексом `j` и с индексом `j+1`. При первом переборе просматриваются все элементы. Последний допустимый индекс в массиве определяется значением `symbs.Length-1`. Из условия, что больший индекс `j+1` принимает максимально возможное значение `symbs.Length-1`, получаем, что максимально возможное значение индекса `j` равно `symbs.Length-2`. За каждый перебор просматривается на один элемент меньше, отсюда и получаем, что индекс `j` в общем случае должен быть меньше, чем `symbs.Length-i`.

В теле внутреннего оператора цикла размещен условный оператор. В условном операторе проверяется условие `symbs[j]>symbs[j+1]`. Оно истинное, если элемент слева больше элемента справа. Если так, то сначала командой `s=symbs[j+1]` значение «элемента справа» записывается в переменную `s`. Командой `symbs[j+1]=symbs[j]` значение «элемента слева» присваивается «элементу справа». Наконец, командой `symbs[j]=s` «элементу слева» присваивается значение переменной `s`, которое есть исходное значение «элемента справа». Таким образом, состоялся обмен значениями между элементами.

Важный момент связан с процедурой сравнения значений элементов массива в условном операторе. Пикантность ситуации в том, что сравнивают символьные значения. В таком случае выполняется сравнение кодов символов. Поэтому сортировка символьного массива в порядке возрастания означает, что значения будут упорядочены от символов с наименьшим значением кода до символов с наибольшим значением кода.

ПОДРОБНОСТИ

В кодовой таблице коды букв следуют один за другим в соответствии с тем, как буквы размещены в алфавите. Код строчных (маленьких) букв больше кодов прописных (больших) букв. Например, код английской буквы `'A'` равен `65`, а код английской буквы `'a'` равен `97`. Русская буква `'A'` в стандартной раскладке имеет код `1040`, а русская буква `'a'` имеет код `1072`. Поэтому при сортировке элементов символьного массива в порядке возрастания сначала следуют большие английские буквы в алфавитном порядке, затем английские маленькие буквы в алфавитном порядке, затем большие русские буквы в алфавитном порядке и после этого — маленькие русские буквы в алфавитном порядке.

После того как сортировка массива закончена, с помощью оператора цикла в консольном окне отображаются значения элементов уже отсортированного массива.

Цикл по массиву

— Товарищ старший лейтенант, только я вас прошу все ценности принять но ониси.
— Ну какой разговор. По усей форме: опись, протокол, сдал-принял, отпечатки пальцев.

из к/ф «Бриллиантовая рука»

Кроме операторов цикла, которые мы рассматривали ранее, есть еще один оператор цикла `foreach`, который называется *циклом по коллекции*. Он может использоваться для перебора элементов в массиве. Синтаксис описания оператора цикла по коллекции такой (жирным шрифтом выделены ключевые элементы шаблона):

```
foreach(тип переменная in массив){
   // Команды
}
```

В круглых скобках после ключевого слова `foreach` объявляется локальная переменная. Она доступна только в теле оператора цикла. При объявлении переменной указывается ее тип и название. После имени переменной указывается ключевое слово `in` и название массива, элементы которого перебираются в процессе выполнения оператора цикла. Команды, выполняемые за каждый цикл, указываются в фигурных скобках после `foreach`-инструкции. Оператор выполняется следующим образом. Объявленная в `foreach`-инструкции переменная последовательно принимает значения элементов массива, указанного после ключевого слова `in`. Таким образом, в отличие от «классического» оператора цикла `for`, в котором элементы массива перебираются с помощью индексной переменной, в данном случае перебираются не индексы элементов, а сами элементы.

ПОДРОБНОСТИ

Важное ограничение состоит в том, что переменная, принимающая значения элементов массива, может быть использована только для считывания значений элементов массива. Через эту переменную нельзя получить доступ к элементу массива для присваивания ему значения.

Небольшой пример использования оператора цикла по коллекции приведен в листинге 4.7.

Листинг 4.7. Использование цикла по коллекции

```
using System;
class ForeachDemo{
   static void Main(){
      // Целочисленный массив:
      int[] nums={1,3,4,8,9};
      // Символьный массив:
      char[] symbs={'a','b','A','B','Ы'};
      // Текстовый массив:
      String[] txts={"красный","желтый","синий"};
      Console.WriteLine("Целочисленный массив");
      // Циклы по целочисленному массиву:
      foreach(int s in nums){
         Console.WriteLine("Число {0} – {1}", s, s%2==0?"четное":"нечетное");
      }
      Console.WriteLine("Символьный массив");
      // Цикл по символьному массиву:
      foreach(char s in symbs){
         Console.WriteLine("Код символа {0} – {1}", s,(int)s);
      }
      Console.WriteLine("Текстовый массив");
      // Цикл по текстовому массиву:
      foreach(string s in txts){
         Console.WriteLine("В слове \"{0}\"{1} букв", s, s.Length);
      }
   }
}
```

Результат выполнения программы таков.

Результат выполнения программы (из листинга 4.7)

```
Целочисленный массив
Число 1 – нечетное
```

```
Число 3 – нечетное
Число 4 – четное
Число 8 – четное
Число 9 – нечетное
Символьный массив
Код символа a – 97
Код символа b – 98
Код символа A – 65
Код символа B – 66
Код символа Ы – 1067
Текстовый массив
В слове "красный" 7 букв
В слове "желтый" 6 букв
В слове "синий" 5 букв
```

В программе создается три массива: целочисленный `nums`, символьный `symbs` и текстовый `txts`. Содержимое каждого из этих массивов отображается в консольном окне. Для перебора элементов в массивах используется оператор цикла `foreach`. В программе таких циклов три (по количеству массивов). В каждом из них использована локальная переменная для перебора элементов массива. Хотя во всех операторах цикла название у локальной переменной `s`, но это разные переменные и объявлены они с разными идентификаторами типов. Тип локальной переменной определяется типом элементов массива. Поэтому в первом операторе цикла переменная `s` объявлена с идентификатором типа `int`, во втором операторе цикла локальная переменная объявлена с идентификатором типа `char`, а в третьем операторе цикла переменная объявлена с идентификатором типа `string`. Соответственно, в первом операторе цикла переменная `s` обрабатывается как целочисленное значение, во втором операторе цикла переменная `s` обрабатывается как символьное значение, а в третьем операторе цикла переменная `s` обрабатывается как текстовое значение. В первом операторе цикла переменная `s` последовательно принимает значения элементов из массива `nums`. За каждый цикл командой `Console.WriteLine("Число {0} – {1}", s, s%2==0?"четное":"нечетное")` отображается сообщение, в котором

указано значение числа и отображен тот факт, четное число или нечетное. В частности, при отображении текстовой строки "Число {0} — {1}" вместо инструкции {0} подставляется значение переменной s (значение очередного элемента массива nums), а вместо инструкции {1} подставляется значение выражения s%2==0?"четное":"нечетное". В выражении использован тернарный оператор. Значением выражения является текст "четное", если истинно условие s%2==0 (остаток от деления на 2 значения элемента массива равен нулю). Если условие s%2==0 ложно, то значением выражения s%2==0?"четное":"нечетное" является текст "нечетное".

Во втором операторе цикла переменная s последовательно принимает значения элементов из массива symbs. Для каждого такого значения отображается сам символ и его код. В теле оператора цикла использована команда Console.WriteLine("Код символа {0} — {1}", s,(int)s). Вместо инструкции {0} подставляется значение элемента массива (оно же значение переменной s), а вместо инструкции {1} подставляется код этого символа. Код символа вычисляется инструкцией (int)s. Здесь имеет место явное приведение символьного значения переменной s к целочисленному формату. Результатом является код соответствующего символа в используемой кодовой таблице.

В третьем операторе цикла переменная s принимает значения элементов из текстового массива txts. Для каждого элемента массива вычисляется количество символов в тексте. Для этого использована инструкция s.Length. Текстовое значение элемента и количество символов в тексте отображаются с помощью команды Console.WriteLine ("В слове \"{0}\"{1} букв", s, s.Length) в теле оператора цикла.

ПОДРОБНОСТИ

При отображении текстового сообщения командой Console.WriteLine("В слове \"{0}\"{1} букв", s, s.Length) мы заключаем значение элемента в двойные кавычки. Другими словами, в текстовый литерал нам необходимо включить двойные кавычки (как символ). В таком случае (то есть для вставки в текстовый литерал двойных кавычек) используется инструкция \", содержащая перед двойной кавычкой " обратную косую черту \.

Двумерные массивы

Ничего особенного. Обыкновенная контрабанда.

из к/ф «Бриллиантовая рука»

В *двумерном массиве* для однозначной идентификации элемента в массиве необходимо указать два индекса. Двумерный массив удобно представлять в виде таблицы. Первый индекс определяет строку, в которой находится элемент, а второй индекс определяет столбец. Создается двумерный массив практически так же, как и одномерный: объявляется переменная массива, создается собственно массив, и ссылка на этот массив записывается в переменную массива. При объявлении переменной двумерного массива после идентификатора типа указываются квадратные скобки с запятой внутри `[,]`. Например, если мы хотим объявить переменную для двумерного целочисленного массива, то соответствующая инструкция могла бы выглядеть как `int[,] nums`.

При создании двумерного массива используется инструкция `new`, а после нее указывается идентификатор типа для элементов массива и в квадратных скобках разделенные через запятую два целых числа, определяющих размер массива по каждому из индексов (количество строк и столбцов в двумерном массиве). Например, создать двумерный целочисленный массив из `3` строк и `5` столбцов можно инструкцией `new int[3,5]`. Как и для одномерного массива, значением инструкции, создающей двумерный массив, является ссылка на этот массив. Поэтому если переменная `nums` предварительно объявлена командой `int[,] nums`, то законной была бы команда `nums=new int[3,5]`. Команды объявления переменной массива и создания массива можно объединять: `int[,] nums=new int[3,5]`.

Ниже приведен шаблон объявления переменной для двумерного массива и создания двумерного массива:

```
тип[,] переменная;
переменная=new тип[размер, размер];
```

Можно использовать и такой шаблон:

```
тип[,] переменная=new тип[размер, размер];
```

Для обращения к элементу массива указывают имя массива и в квадратных скобках через запятую — индексы этого элемента. Индексация по каждому индексу начинается с нуля. То есть инструкция `nums[0,0]` означает обращение к элементу массива `nums`, который расположен в первой строке и первом столбце (в строке с индексом `0` и столбце с индексом `0`). Выражение `nums[1,2]` является обращением к элементу, находящемуся в строке с индексом `1` (вторая по порядку строка) и столбце с индексом `2` (третий по порядку столбец).

Свойство `Length` для двумерного массива возвращает общее количество элементов в массиве. То есть для массива `nums` из `3` строк и `5` столбцов значением выражения `nums.Length` является число `15` (произведение `3` на `5`). Чтобы узнать размер массива по какому-то индексу (то есть количество строк или количество столбцов в массиве), используют метод `GetLength()`. Метод вызывается из переменной массива, а аргументом методу передается целое число, определяющее индекс, для которого возвращается размер массива.

ПОДРОБНОСТИ

Массивы в C# по факту реализуются как объекты. Как и у прочих объектов, у массива есть методы (и свойства). Более детально объекты обсуждаются немного позже. Здесь для нас важно научиться пользоваться преимуществами, которые имеются у массивов благодаря такому «объектному» способу их реализации.

Размерность массива определяется количеством индексов, которые нужно указать для идентификации элемента массива. Определить размерность массива можно с помощью свойства `Rank`. Размерность двумерного массива равна `2`.

Если мы вызываем метод `GetLength()` с аргументом `0`, то значением возвращается размер массива по первому индексу. Если вызвать метод `GetLength()` с аргументом `1`, то получим размер массива по второму индексу.

Например, если массив `nums` состоит из `3` строк и `5` столбцов, то значением выражения `nums.GetLength(0)` является число `3` (размер по первому индексу — количество строк), а значением выражения `nums.GetLength(1)` является число `5` (размер по второму индексу — количество столбцов).

В листинге 4.8 приведен пример, в котором иллюстрируется способ создания двумерного массива: создается целочисленный массив и построчно заполняется натуральными числами.

Листинг 4.8. Двумерный массив

```
using System;
class TwoDimArrayDemo{
   static void Main(){
      // Количество строк и столбцов в массиве:
      int rows=3, cols=5;
      // Создание двумерного массива:
      int[,] nums=new int[rows, cols];
      // Значение первого элемента в массиве:
      int value=1;
      // Заполнение и отображение массива.
      // Перебор строк в массиве:
      for(int i=0; i<nums.GetLength(0); i++){
         // Перебор столбцов в строке:
         for(int j=0; j<nums.GetLength(1); j++){
            // Присваивание значения элементу массива:
            nums[i, j]=value;
            // Это будет значение следующего элемента:
            value++;
            // Отображение элемента в строке:
            Console.Write(nums[i, j]+"\t");
         }
         // Переход к новой строке:
         Console.WriteLine();
      }
   }
}
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 4.8)

```
1       2       3       4       5
6       7       8       9       10
11      12      13      14      15
```

В программе количество строк и столбцов в двумерном массиве задается через целочисленные переменные `rows` и `cols` (значения `3` и `5` соответственно). Двумерный массив создается командой `int[,] nums=new int[rows, cols]`. Для заполнения массива и отображения значений его элементов используются вложенные операторы цикла. Во внешнем операторе цикла индексная переменная `i` перебирает строки двумерного массива. Начальное значение переменной равно `0`, а верхняя граница определяется значением выражения `nums.GetLength(0)` (количество строк в массиве `nums` — переменная `i` строго меньше этого значения). Во внутреннем операторе цикла переменная `j` перебирает столбцы двумерного массива. Начальное значение переменной равно `0`, и за каждый цикл переменная увеличивается на единицу. Оператор цикла выполняется, пока значение переменной строго меньше значения выражения `nums.GetLength(1)` (количество столбцов в массиве `nums`). При заданных значениях индексов `i` и `j` командой `nums[i, j]=value` соответствующему элементу массива присваивается значение. Начальное значение переменной `value` равно `1`, поэтому первый элемент в массиве (элемент с двумя нулевыми индексами — элемент в первой строке в первом столбце) получает единичное значение. После выполнения команды `value++` значение переменной `value` увеличивается на единицу. На следующем цикле это новое значение будет присвоено очередному элементу.

После того как значение элементу присвоено, командой `Console.Write(nums[i, j]+"\t")` оно отображается в консольном окне. Элементы из одной строки массива отображаются в одной и той же строке консольного окна. Для размещения элементов в строке использована табуляция.

Тело внешнего оператора цикла состоит из внутреннего оператора цикла и команды `Console.WriteLine()`. Внутренний оператор цикла нужен для заполнения строк массива и отображения значений элементов в консольном окне. Когда внутренний оператор цикла завершает работу, благодаря команде `Console.WriteLine()` выполняется переход к новой строке. На этом завершается очередной цикл внешнего оператора цикла. На следующем цикле внешнего оператора цикла снова выполняется внутренний оператор цикла, и затем команда `Console.WriteLine()`, и так далее, пока не будут перебраны все строки двумерного массива.

Как и одномерный массив, двумерный массив можно инициализировать при создании. В этом случае переменной массива присваивается

список со значениями, которыми инициализируется двумерный массив. Список значений заключается в фигурные скобки. Его элементами являются вложенные списки. Каждый вложенный список выделяется парой фигурных скобок. Значения, указанные во вложенных списках, определяют строки двумерного массива. Например, командой `int[,] nums={{1,2,3},{4,5,6}}` создается целочисленный двумерный массив из двух строк и трех столбцов. Элементы в первой строке принимают значения 1, 2 и 3, а элементы во второй строке принимают значения 4, 5 и 6. А если воспользоваться командой `int[,] nums={{1,2},{3,4},{5,6}}`, то получим массив из трех строк и двух столбцов (в первой строке — элементы со значениями 1 и 2, во второй строке — элементы со значениями 3 и 4, в третьей строке — элементы со значениями 5 и 6).

ПОДРОБНОСТИ

Наряду с командой `int[,] nums={{1,2,3},{4,5,6}}` для инициализации массива могут использоваться и команды `int[,] nums=new int[2,3]{{1,2,3},{4,5,6}}` или `int[,] nums=new int[,]{{1,2,3},{4,5,6}}`. Если размеры массива не указаны, они определяются автоматически по списку инициализации. Если размеры массива указаны явно, то они должны соответствовать структуре списка инициализации.

Далее рассматривается пример, в котором, кроме прочего, используется инициализация двумерного массива. В символьный массив из двух строк и трех столбцов вставляется строка и столбец. Индексы добавляемых строки и столбца определяются с помощью генератора случайных чисел. Чтобы понять, как это делается, рассмотрим программный код в листинге 4.9.

Листинг 4.9. Добавление строки и столбца в массив

```
using System;
class InitTwoDimArrayDemo{
   static void Main(){
      // Создание и инициализация двумерного массива:
      char[,] symbs={{'A','B','C'},{'D','E','F'}};
      Console.WriteLine("Исходный массив:");
      // Отображение массива:
```

```
for(int i=0; i<symbs.GetLength(0); i++){
   for(int j=0; j<symbs.GetLength(1); j++){
      // Отображение значения элемента:
      Console.Write(symbs[i, j]+" ");
   }
   // Переход к новой строке:
   Console.WriteLine();
}
// Объект для генерирования случайных чисел:
Random rnd=new Random();
// Строка и столбец:
int row=rnd.Next(symbs.GetLength(0)+1);
int col=rnd.Next(symbs.GetLength(1)+1);
Console.WriteLine("Добавляется {0}-я строка и {1}-й столбец", row, col);
// Создание нового массива:
char[,] tmp=new char[symbs.GetLength(0)+1, symbs.GetLength(1)+1];
// Целочисленные переменные:
int a, b;
// Символьная переменная:
char s='a';
// Заполнение массива. Копирование значений
// из исходного массива:
for(int i=0; i<symbs.GetLength(0); i++){
   // Первый индекс для элемента нового массива:
   if(i<row) a=i;
   else a=i+1;
   for(int j=0; j<symbs.GetLength(1); j++){
      // Второй индекс для элемента нового массива:
      if(j<col) b=j;
      else b=j+1;
      // Присваивание значения элементу массива:
      tmp[a, b]=symbs[i, j];
```

```
            }
        }
        // Заполнение добавленной строки в новом массиве:
        for(int j=0; j<tmp.GetLength(1); j++){
            // Значение элемента в строке:
            tmp[row, j]=s;
            // Новое значение для следующего элемента:
            s++;
        }
        for(int i=0; i<tmp.GetLength(0); i++){
            // Если элемент не в добавленной строке:
            if(i!=row){
                // Значение элемента в столбце:
                tmp[i, col]=s;
                // Новое значение для следующего элемента:
                s++;
            }
        }
        // Присваивание массивов:
        symbs=tmp;
        Console.WriteLine("Новый массив:");
        // Отображение массива:
        for(int i=0; i<symbs.GetLength(0); i++){
            for(int j=0; j<symbs.GetLength(1); j++){
                // Отображение значения элемента:
                Console.Write(symbs[i, j]+" ");
            }
            // Переход к новой строке:
            Console.WriteLine();
        }
    }
}
```

С поправкой на использование генератора случайных чисел результат выполнения программы может быть таким.

Результат выполнения программы (из листинга 4.9)

```
Исходный массив:
A B C
D E F
Добавляется 1-я строка и 2-й столбец
Новый массив:
A B д C
а б в г
D E e F
```

Или таким.

Результат выполнения программы (из листинга 4.9)

```
Исходный массив:
A B C
D E F
Добавляется 2-я строка и 3-й столбец
Новый массив:
A B C д
D E F e
а б в г
```

Или, например, таким.

Результат выполнения программы (из листинга 4.9)

```
Исходный массив:
A B C
D E F
Добавляется 0-я строка и 0-й столбец
Новый массив:
а б в г
```

```
д A B C
e D E F
```

Есть и другие варианты. Мы же проанализируем программный код.

Исходный символьный массив `symbs` создается и инициализируется командой `char[,] symbs={{'A','B','C'},{'D','E','F'}}`. Это массив из двух строк и трех столбцов, заполненный английскими буквами в алфавитном порядке (хотя в данном случае это не принципиально). С помощью вложенных операторов цикла содержимое этого массива отображается в консольном окне. Поскольку нам понадобится генерировать случайные числа, командой `Random rnd=new Random()` создается объект `rnd` класса `Random`. Объект используется для определения индекса добавляемой строки и индекса добавляемого столбца. Индекс строки записывается в переменную `row`, а значение вычисляется как `rnd.Next(symbs.GetLength(0)+1)`. Это случайное число в диапазоне от `0` до `symbs.GetLength(0)` (верхняя допустимая граница определяется количеством строк в исходном массиве). Индекс добавляемого столбца записывается в переменную `col` и вычисляется командой `rnd.Next(symbs.GetLength(1)+1)`. Это случайное число в диапазоне от `0` до `symbs.GetLength(1)` (верхняя допустимая граница определяется количеством столбцов в исходном массиве). После того как индексы добавляемой строки и столбца определены, командой `Console.WriteLine("Добавляется {0}-я строка и {1}-й столбец", row, col)` отображается сообщение об индексе добавляемой строки и индексе добавляемого столбца.

Командой `char[,] tmp=new char[symbs.GetLength(0)+1, symbs.GetLength(1)+1]` создается новый символьный массив. В нем на одну строку и на один столбец больше, чем в исходном массиве `symbs`. После создания массив `tmp` заполняется. Делается это в несколько этапов. Сначала в массив `tmp` копируются значения из исходного массива `symbs`. Но специфика этого процесса состоит в том, что если первый индекс элемента в массиве `symbs` больше или равен индексу `row` добавляемой строки, то при копировании нужно выполнить сдвиг на одну позицию вниз. Аналогично, если второй индекс элемента в массиве `symbs` больше или равен индексу `col` добавляемого столбца, то при копировании необходимо выполнить сдвиг на одну позицию вправо. Другими словами, индекс копируемого элемента в массиве `symbs` и индекс элемента в массиве `tmp`, которому присваивается

значение, могут отличаться. С помощью вложенных операторов цикла перебираются элементы в массиве symbs. Индексы элемента из массива tmp, которому присваивается значение, записываются в переменные a и b. Во внешнем операторе цикла (индексная переменная i перебирает строки в массиве symbs) использован условный оператор, которым переменой a присваивается значение i при условии i<row и i+1 в противном случае. Во внутреннем операторе цикла (индексная переменная j перебирает столбцы в массиве symbs) с помощью условного оператора переменной b присваивается значение j при условии j<col и значение j+1 в противном случае. Командой tmp[a, b]=symbs[i, j] присваивается значение элементу в массиве tmp.

После выполнения вложенных операторов цикла в массиве tmp присвоены значения всем элементам, за исключением тех, что находятся в добавляемой строке и добавляемом столбце. Эти строка и столбец заполняются отдельно. Сначала заполняется добавляемая строка: запускается оператор цикла, в котором индексная переменная j перебирает элементы в строке с индексом row. Значение элементу в строке присваивается командой tmp[row, j]=s. Начальное значение переменной s равно 'a'(маленькая буква русского алфавита). Такое значение будет у первого элемента в добавленной строке. Для следующего элемента значением будет следующая буква в алфавите, поскольку мы использовали команду s++, которой значение переменной s «увеличивается на единицу»: значением переменной становится следующий символ в алфавите.

После заполнения добавленной строки заполняется добавленный столбец. В соответствующем операторе цикла индексная переменная i перебирает элементы в столбце с индексом col. Значение элементам присваивается командой tmp[i, col]=s. Единственное, это происходит лишь в случае, если элемент не находится в добавленной строке (значение этого элемента уже определено при заполнении строки). Поэтому мы использовали условный оператор с условием i!=row (первый индекс i элемента не совпадает с индексом row добавленной строки).

На этом заполнение массива tmp завершается. Но мы идем дальше и выполняем присваивание массивов командой symbs=tmp. В результате и переменная symbs, и переменная tmp ссылаются на один и тот же массив — это новый массив, созданный путем добавления к исходному массиву строки и столбца. Для проверки содержимого созданного массива использованы вложенные операторы цикла.

НА ЗАМЕТКУ

Мы создали новый массив и ссылку в переменной массива `symbs` «перебросили» на этот новый массив. В итоге создается иллюзия, что изменен исходный массив. Так, вложенные операторы цикла в самом начале программы и в самом конце программы фактически одинаковы. Эти блоки команд используются для отображения содержимого массива, на который ссылается переменная `symbs`. Но массивы отображаются разные. Причина в том, что в начале выполнения программы и в конце выполнения программы переменная `symbs` ссылается на разные массивы. Причем у них не просто разные значения элементов, а разное количество строк и столбцов.

Что касается исходного массива, на который ранее ссылалась переменная `symbs`, то, поскольку на этот массив больше не ссылается ни одна переменная, массив из памяти будет удален системой сборки мусора.

Многомерные массивы

> Три магнитофона, три кинокамеры заграничных, три нортсигара отечественных, куртка замшевая... Три куртки.
>
> *из к/ф «Иван Васильевич меняет профессию»*

Размерность массива может быть большей двух. Правда, на практике массивы размерности большей, чем два, используются достаточно редко. Но это вопрос другой.

НА ЗАМЕТКУ

Напомним, что размерность массива определяется количеством индексов, которые следует указать для однозначной идентификации элемента массива.

Итак, многомерные массивы реализуются таким же образом, как одномерные и двумерные массивы: нам нужна переменная массива и, собственно, сам массив. Ссылка на массив записывается в переменную массива. Поэтому, как и во всех предыдущих случаях (с одномерными и двумерными массивами), нам предстоит объявлять переменную массива, создавать сам массив и записывать в переменную массива ссылку на массив.

Переменная для многомерного массива объявляется следующим образом: указывается идентификатор типа для элементов массива, после которого следуют квадратные скобки с запятыми внутри. Количество запятых на единицу меньше размерности массива. Скажем, если мы объявляем переменную для массива размерности 5, то внутри квадратных скобок должны быть 4 запятые. После квадратных скобок с запятыми указывается имя переменной массива. Значением этой переменной можно присвоить ссылку на массив. Он должен быть той же размерности, которая указана при создании переменной массива, а тип элементов массива должен совпадать с типом, указанным при объявлении переменной массива. Например, командой `int[,,] nums` объявляется переменная `nums` для трехмерного целочисленного массива, а командой `char[,,,,] symbs` объявляется переменная `symbs` для символьного массива размерности `5`.

Многомерный массив создается с помощью оператора `new`. После оператора `new` указывается идентификатор типа элементов массива, и в квадратных скобках перечисляются размеры массива по каждому из индексов. Значением инструкции по созданию массива является ссылка на массив. Эта ссылка может быть записана в переменную массива. Например, командой `int[,,] nums=new int[3,4,5]` создается трехмерный целочисленный массив размерами `3` на `4` на `5`.

НА ЗАМЕТКУ

Одномерный массив можно представлять в виде цепочки элементов. Двумерный массив удобно представлять в виде таблицы. Трехмерный массив можно представить как сложенный из элементов прямоугольный параллелепипед. Первый индекс определяет «длину», второй определяет «ширину», и третий определяет «высоту», на которой элемент находится в «параллелепипеде». С массивами больших размерностей геометрическая интерпретация не так очевидна.

При создании многомерных массивов можно использовать инициализацию аналогично тому, как мы это делали для одномерных и двумерных массивов. Значением переменной массива присваивается список, элементами которого являются списки, элементами которых также являются списки, и так далее — количество вложений определяется размерностью массива. Например, командой `int[,,] nums={{{0,0,0,0},{0,0,0,1},{0,0,1,0}},{{0,0,1,1},{0,1,0,0},{0,1,0,1}}}` создается трехмерный целочисленный массив, который:

- по первому индексу имеет размер 2 (поскольку внешний список содержит два внутренних списка);
- по второму индексу массив имеет размер 3 (каждый внутренний список содержит по 3 вложенных списка-элемента);
- по третьему индексу размер массива равен 4 (поскольку каждый вложенный список-элемент содержит по 4 значения).

Мы также могли бы использовать команды `int[,,] nums=new int[2,3,4]{{{0,0,0,0},{0,0,0,1},{0,0,1,0}},{{0,0,1,1},{0,1,0,0},{0,1,0,1}}}` или `int[,,] nums=new int[,,]{{{0,0,0,0},{0,0,0,1},{0,0,1,0}},{{0,0,1,1},{0,1,0,0},{0,1,0,1}}}`.

Скромный пример, в котором использован многомерный (трехмерный, если точнее) символьный массив, представлен в листинге 4.10.

Листинг 4.10. Знакомство с многомерными массивами

```
using System;
class MultiDimArrayDemo{
   static void Main(){
      // Трехмерный символьный массив:
      char[,,] symbs=new char[4,3,5];
      // Одномерный символьный массив:
      char[] s={'б','Б','b','B'};
      // Заполнение трехмерного массива и отображение
      // значений его элементов.
      // Цикл по первому индексу:
      for(int i=0; i<symbs.GetLength(0); i++){
         // Отображение сообщения:
         Console.WriteLine(" Слой № {0}:", i+1);
         // Цикл по второму индексу:
         for(int j=0; j<symbs.GetLength(1); j++){
            // Цикл по третьему индексу:
            for(int k=0; k<symbs.GetLength(2); k++){
               // Присваивание значения элементу массива:
```

```
                symbs[i, j, k]=s[i];
                // Изменение элемента в одномерном массиве:
                s[i]++;
                // Отображение значения элемента массива:
                Console.Write(symbs[i, j, k]+" ");
            }
            // Переход к новой строке:
            Console.WriteLine();
        }
        // Отображение "линии":
        Console.WriteLine("----------");
    }
  }
}
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 4.10)

```
Слой № 1:
б в г д е
ж з и й к
л м н о п
----------
Слой № 2:
Б В Г Д Е
Ж З И Й К
Л М Н О П
----------
Слой № 3:
b c d e f
g h i j k
l m n o p
----------
```

```
 Слой № 4:
B C D E F
G H I J K
L M N O P
----------
```

Командой `char[,,] symbs=new char[4,3,5]` создается трехмерный символьный массив. Размер массива по первому индексу равен `4`, размер массива по второму индексу равен `3`, и размер массива по третьему индексу равен `5`. Этот массив можно представить себе как «слойку» из четырех таблиц, каждая из трех строк и пяти столбцов. Мы эти таблицы заполняем символами — буквами алфавита. Мы используем русский и английский алфавит, маленькие и большие буквы — то есть всего четыре варианта, по количеству таблиц в «слойке». Для заполнения трехмерного массива используется конструкция из трех вложенных операторов цикла. Первый внешний цикл с индексной переменной `i` выполняется по первому индексу элементов массива. Для перебора второго индекса использован вложенный оператор цикла с индексной переменной `j`, а третий индекс перебирается с помощью оператора цикла с индексной переменной `k`. Для определения размера массива по первому индексу использована инструкция `symbs.GetLength(0)`, инструкцией `symbs.GetLength(1)` вычисляется размер массива по второму индексу, а размер по третьему индексу определяется выражением `symbs.GetLength(2)`. Значение элементу массива присваивается командой `symbs[i, j, k]=s[i]`. Здесь `s[i]` — это элемент одномерного символьного массива из четырех элементов (в соответствии с размером трехмерного массива `symbs` по первому индексу). Массив `s` содержит буквы, с которых начинается «послойное» заполнение трехмерного массива. После присваивания значения элементу `s[i, j, k]` значение в одномерном символьном массиве изменяется командой `s[i]++`, так что у следующего элемента значение будет другим. После присваивания значения элементу трехмерного массива оно выводится в консольное окно.

Массив со строками разной длины

Ох, красота-то какая! Лепота!

из к/ф «Иван Васильевич меняет профессию»

Напомним, как создается одномерный массив: объявляется переменная массива, и значением ей присваивается ссылка на массив. Впоследствии переменная массива отождествляется с массивом, на который она ссылается. Важно здесь то, что такую переменную можно индексировать — после имени переменной в квадратных скобках указывается индекс, в результате чего мы получаем доступ к элементу массива. А теперь представим, что мы создали одномерный массив, элементами которого являются переменные массива. Общая схема будет следующей: есть переменная массива, которая ссылается на массив, и этот массив состоит из переменных массива, ссылающихся на массивы. Именно такой подход реализован в примере, представленном в листинге 4.11.

Листинг 4.11. Массив из переменных массива

```
using System;
class AnotherTwoDimArrayDemo{
   static void Main(){
      // Символьный массив из переменных массива:
      char[][] symbs=new char[5][];
      // Целочисленный массив из переменных массива:
      int[][] nums={new int[]{1,2,3}, new int[]{4,5}, new int[]{6,7,8,9}};
      // Символьная переменная:
      char s='A';
      // Заполнение символьного массива.
      // Перебор элементов во внешнем массиве:
      for(int i=0; i<symbs.Length; i++){
         // Создание внутреннего массива:
         symbs[i]=new char[i+3];
         // Перебор элементов во внутреннем массиве:
         for(int j=0; j<symbs[i].Length; j++){
            // Значение элемента внутреннего массива:
```

```
            symbs[i][j]=s;
            // Значение для следующего элемента:
            s++;
         }
      }
      Console.WriteLine("Целочисленный массив:");
      // Отображение целочисленного массива:
      for(int i=0; i<nums.Length; i++){
         for(int j=0; j<nums[i].Length; j++){
            // Отображение элемента массива:
            Console.Write("{0,3}", nums[i][j]);
         }
         Console.WriteLine();
      }
      Console.WriteLine("Символьный массив:");
      // Отображение символьного массива.
      // Перебор элементов внешнего массива:
      foreach(char[] q in symbs){
         // Перебор элементов во внутреннем массиве:
         foreach(char p in q){
            // Отображение элемента массива:
            Console.Write("{0,2}", p);
         }
         Console.WriteLine();
      }
   }
}
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 4.11)

```
Целочисленный массив:
  1  2  3
```

```
  4  5
  6  7  8  9
Символьный массив:
 А Б В
 Г Д Е Ж
 З И Й К Л
 М Н О П Р С
 Т У Ф Х Ц Ч Ш
```

В программе есть несколько моментов, достойных внимания. Итак, командой `char[][] symbs=new char[5][]` создается массив `symbs`. Но это не очень обычный массив. Это массив одномерный, не двумерный. Массив из пяти элементов. Элементами массива являются переменные, которые могут ссылаться на символьные одномерные массивы. При объявлении переменной массива `symbs` после идентификатора типа `char` использованы двойные прямоугольные скобки `[][]`. Одна пара квадратных скобок означает, что создается массив. Вторая относится непосредственно к идентификатору типа. Фактически получается, что мы объявляем переменную для одномерного массива, который состоит из элементов типа `char[]`. А такое выражение используется при объявлении переменной для символьного массива. Далее, инструкция `new char[5][]` означает, что создается одномерный массив из пяти элементов (значение `5` в первых квадратных скобках). Вторые пустые квадратные скобки относятся к идентификатору типа `char`. Получается, что мы создаем массив из элементов типа `char[]` и размер массива равен `5`. Итог такой: переменная `symbs` ссылается на массив из пяти элементов. Каждый элемент массива является переменной массива. Каждая такая переменная может ссылаться на одномерный символьный массив (но пока не ссылается, поскольку самих массивов еще нет — их нужно создать). Массивы создаются и заполняются с помощью операторов цикла. С помощью оператора цикла с индексной переменной `i` перебираются элементы в массиве `symbs` (будем называть его внешним массивом). Индексная переменная `i` принимает значения в диапазоне, ограниченном сверху величиной `symbs.Length` (условие *«строго меньше»*). Здесь еще раз имеет смысл подчеркнуть, что массив `symbs` — одномерный. В теле этого оператора цикла командой `symbs[i]=new char[i+3]` создается символьный массив, и ссылка на него записывается в элемент `symbs[i]`. Размер массива определяется значением выражения `i+3`. То есть массивы, создаваемые в рамках разных циклов,

имеют разные размеры. И эти массивы нужно заполнять. Поэтому используется внутренний оператор цикла с индексной переменной `j`. Верхняя граница диапазона изменения индексной переменной `j` определяется выражением `symbs[i].Length` (условие *«строго меньше»*). Здесь мы должны учесть, что `symbs[i]` — это переменная, которая ссылается на одномерный массив, и фактически переменная обрабатывается как одномерный массив. Значением выражения `symbs[i].Length` является количество элементов в массиве, на который ссылается переменная `symbs[i]`.

В теле внутреннего оператора цикла командой `symbs[i][j]=s` присваивается значение элементу во внутреннем массиве (массив, на который ссылается переменная `symbs[i]`). Начальное значение переменной `s` равно `'A'` (большая буква русского алфавита). За каждый цикл значение переменной `s` «увеличивается на единицу», в результате чего элементы массива заполняются большими буквами русского алфавита.

Для отображения элементов созданного «массива из массивов» использован оператор цикла по коллекции. Точнее, мы используем вложенные операторы цикла `foreach`.

Во внешнем операторе `foreach` перебираются элементы массива `symbs`, которые, как отмечалось выше, сами являются массивами (переменными, ссылающимися на массивы). Локальная переменная `q` во внешнем операторе цикла указана с идентификатором типа `char[]` и как такая, что принимает значения элементов массива `symbs`. Значение переменной `q` — это ссылка на массив. Поэтому переменная `q` обрабатывается как массив. Как массив она использована во внутреннем операторе цикла `foreach`. Там объявлена локальная переменная `p` типа `char`, и эта переменная последовательно принимает значения элементов массива `q`. Таким образом, если `q` является элементом массива `symbs`, то `p` — это элемент в массиве, на который ссылается элемент массива `symbs`. Значение этого элемента отображается командой `Console.Write("{0,2}", p)`. Первый `0` в инструкции `{0,2}` означает, что вместо нее подставляется значение первого аргумента после текстовой строки `"{0,2}"` (то есть значение переменной `p`), а число `2` в инструкции `{0,2}` означает, что для отображения значения выделяется не меньше двух позиций.

Программа содержит пример создания еще одного массива, состоящего из переменных массива. На этот раз создается массив, состоящий из ссылок на целочисленные массивы. Особенность в том, что при создании массив инициализируется. Мы использовали команду `int[][]`

`nums={new int[]{1,2,3}, new int[]{4,5}, new int[]{6,7,8,9}}`. Переменная массива `nums` объявлена с идентификатором типа `int[][]`. Это означает, что `nums` может ссылаться на одномерный массив, элементы которого имеют тип `int[]` — то есть являются ссылками на целочисленные массивы. Значением переменной `nums` присваивается список. Список определяет массив, на который будет ссылаться переменная `nums`. В списке всего три элемента: `new int[]{1,2,3}`, `new int[]{4,5}` и `new int[]{6,7,8,9}`. В итоге получается, что переменная `nums` ссылается на массив, состоящий из трех элементов. Первый элемент `nums[0]` ссылается на целочисленный массив, состоящий из элементов со значениями 1, 2 и 3. Второй элемент `nums[1]` ссылается на целочисленный массив, состоящий из элементов со значениями 4 и 5. Третий элемент `nums[2]` ссылается на целочисленный массив, состоящий из элементов со значениями 6, 7, 8 и 9. Как так получилось? Нужно вспомнить, что элементы массива `nums` — ссылки на одномерные массивы. Но кроме самих ссылок, нужны еще и сами одномерные массивы. Эта «двойная» задача решается с помощью инструкций `new int[]{1,2,3}`, `new int[]{4,5}` и `new int[]{6,7,8,9}`. Например, что такое инструкция `new int[]{1,2,3}`? Это команда создания целочисленного массива с элементами 1, 2 и 3. То есть одна задача решается — создается одномерный целочисленный массив. У инструкции `new int[]{1,2,3}` есть значение — это ссылка на созданный массив. Аналогично с двумя другими инструкциями. Следовательно, получается, что в списке, присваиваемом переменной `nums`, три элемента, которые являются ссылками на одномерные массивы. Эти три значения определяют содержимое массива, на который будет ссылаться переменная `nums`.

ПОДРОБНОСТИ

При формировании списка инициализации для переменной массива `nums` мы не можем использовать список из списков, поскольку такой способ используется при создании двумерного массива, когда каждый внутренний список определяет строку в двумерном массиве. Именно поэтому перед списками, определяющими одномерные целочисленные массивы, использована инструкция `new int[]`.

Для отображения содержимого массива `nums` мы используем обычный оператор цикла `for` (вложенные операторы цикла). Во внешнем операторе цикла индексная переменная `i` (с начальным нулевым значением) перебирает индексы элементов в массиве `nums`. Значение индексной

переменной каждый раз увеличивается на единицу, а цикл выполняется, пока значение переменной меньше значения `nums.Length` (количество элементов в одномерном массиве `nums`). Во внутреннем операторе цикла индексная переменная `j` перебирает элементы в массиве, на который ссылается элемент `nums[i]` массива `nums`. Количество элементов в массиве `nums[i]` дается выражением `nums[i].Length`. За каждый цикл выполняется команда `Console.Write("{0,3}", nums[i][j])`, которой в консольном окне отображается значение элемента `nums[i][j]`, причем под отображаемое число выделяется не менее трех позиций (число `3` после запятой в инструкции `{0,3}`).

Массив объектных ссылок

> Я артист больших и малых академических театров.
>
> *из к/ф «Иван Васильевич меняет профессию»*

Ранее мы всегда явно или неявно имели в виду, что элементы массива относятся к одному типу. В общем-то, это так. Но бывают и неординарные ситуации. Как мы узнаем немного позже, в языке C# в пространстве имен `System` есть один особый класс, который называется `Object`. Его «особенность» связана с тем, что переменная типа `Object` может ссылаться на значение любого типа. Проще говоря, если объявить переменную и идентификатором типа для нее указать имя класса `Object`, то значением такой переменной можно присвоить практически все, что угодно.

ПОДРОБНОСТИ

В языке C# существует механизм наследования, который позволяет создавать один класс (производный класс) на основе другого класса (базовый класс). В результате получается иерархия классов, связанных наследованием, когда класс создается на основе класса, и на его основе создается еще один класс, и так далее. В наследовании есть важный момент: переменная базового класса может ссылаться на объекты, созданные на основе производных классов. Специфика класса `Object` состоит в том, что он находится в вершине иерархии наследования и любой класс является его прямым или косвенным потомком. Поэтому переменная типа `Object` может ссылаться на любой объект. Более подробно методы работы с классами и объектами, в том числе и механизм наследования, описываются в последующих главах книги.

Если создать массив из элементов типа `Object`, то элементам такого массива можно присваивать значения разных типов. В некоторых случаях это бывает удобно и полезно.

НА ЗАМЕТКУ

На практике вместо названия класса `Object` обычно используют идентификатор `object`, который является псевдонимом для выражения `System.Object`.

Небольшой пример, в котором создается и используется массив с элементами типа `Object`, приведен в листинге 4.12.

Листинг 4.12. Массив из переменных типа `Object`

```
using System;
class ObjectArrayDemo{
   static void Main(){
      // Массив из трех переменных типа Object:
      Object[] objs=new Object[3];
      // Элементам массива присваиваются значения
      // разных типов:
      objs[0]=123;              // Целое число
      objs[1]='A';              // Символ
      objs[2]="Третий элемент"; // Текст
      Console.WriteLine("Создан такой массив:");
      // Проверка содержимого массива:
      for(int k=0; k<objs.Length; k++){
         Console.WriteLine(k+": "+objs[k]);
      }
      // Новые значения элементов:
      objs[0]=(int)objs[0]+111; // Целое число
      objs[1]="Второй элемент"; // Текст
      objs[2]=3.141592;         // Действительное число
      Console.WriteLine("После присваивания значений:");
      // Проверка содержимого массива:
```

```
      for(int k=0; k<objs.Length; k++){
         Console.WriteLine(k+": "+objs[k]);
      }
      // Целочисленный массив:
      int[] nums={10,20,30};
      // Переменная массива присваивается как значение
      // элементу массива:
      objs[2]=nums;
      Console.WriteLine("Целочисленный массив:");
      // Отображение элементов целочисленного массива:
      for(int i=0; i<((int[])objs[2]).Length; i++){
         Console.Write("{0,3}",((int[])objs[2])[i]);
      }
      Console.WriteLine();
      // Новое значение элемента в числовом массиве:
      ((int[])objs[2])[1]=0;
      Console.WriteLine("Еще раз тот же массив:");
      // Отображение элементов целочисленного массива:
      for(int i=0; i<nums.Length; i++){
         Console.Write("{0,3}", nums[i]);
      }
      Console.WriteLine();
   }
}
```

Результат выполнения программы будет таким, как показано ниже.

Результат выполнения программы (из листинга 4.12)

```
Создан такой массив:
0: 123
1: A
2: Третий элемент
После присваивания значений:
0: 234
```

```
1: Второй элемент
2: 3,141592
Целочисленный массив:
 10 20 30
Еще раз тот же массив:
 10 0 30
```

Массив из трех переменных типа `Object` создается с помощью команды `Object[] objs=new Object[3]`. После создания массива его элементам присваиваются значения. Делается это в явном виде, но пикантность ситуации в том, что присваиваемые значения относятся к разным типам. Так, командой `objs[0]=123` первому элементу массива `objs` значением присваивается целочисленный литерал `123`. Командой `objs[1]='A'` второму элементу массива `objs` присваивается символьное значение `'A'`, а командой `objs[2]="Третий элемент"` третьему элементу массива `objs` присваивается текстовое значение `"Третий элемент"`. С помощью оператора цикла убеждаемся, что значения элементов именно такие.

На следующем этапе элементам массива присваиваются новые значения. Причем тип присваиваемых значений может не совпадать с типом текущего значения элемента. Здесь все просто. Некоторых пояснений требует лишь команда `objs[0]=(int)objs[0]+111`. Все дело в том, что на момент выполнения команды фактическим значением элемента `objs[0]` является целое число. Что мы пытаемся сделать — так это к текущему значению элемента прибавить число `111`. И вот здесь сталкиваемся с ситуацией, которая является платой за возможность присваивать переменным типа `Object` значения разных типов: если мы хотим, чтобы с элементом `objs[0]` выполнялась арифметическая операция и сам элемент обрабатывался как целочисленное значение, то необходимо этот факт в явном виде отобразить. Именно поэтому в правой части выражения перед ссылкой на элемент `objs[0]` использована инструкция явного приведения типа `(int)`. Фактически, используя выражение вида `(int)objs[0]`, мы даем команду обрабатывать значение элемента `objs[0]` как целое число типа `int`.

После изменения значений элементов массива `objs` мы опять проверяем содержимое массива (с помощью оператора цикла) и убеждаемся, что изменения вступили в силу.

На следующем этапе командой `int[] nums={10,20,30}` создается целочисленный массив из трех элементов со значениями `10`, `20` и `30`. Это обычный массив. Не очень обычен способ его использования: командой `objs[2]=nums` переменную массива `nums` мы присваиваем значением элементу `objs[2]`. Значение переменной `nums` — это ссылка на массив из трех чисел. После выполнения команды `objs[2]=nums` значением элемента `objs[2]` является ссылка на точно тот же массив. Поэтому мы вполне можем использовать для доступа к массиву элемент `objs[2]`. Но если так, то мы должны в явном виде указать, как следует обрабатывать значение элемента `objs[2]` — используется инструкция приведения типа `(int[])`, означающая, что значение элемента является ссылкой на целочисленный массив. Так мы поступаем в операторе цикла, в котором индексная переменная `i` принимает значения от `0` и до некоторого значения: переменная `i` должна быть строго меньше значения `((int[])objs[2]).Length`, которое является размером массива, на который ссылается элемент `objs[2]`. Выражение `((int[])objs[2]).Length` — это значение свойства `Length` для элемента `objs[2]`, если значение элемента интерпретировать как ссылку на одномерный целочисленный массив (значение типа `int[]`). В теле оператора цикла ссылка на элементы массива, на который ссылается элемент `objs[2]`, выполняется в формате `((int[])objs[2])[i]`: сначала значение элемента `objs[2]` приводится к типу `int[]`, а затем то, что получилось, индексируется (в квадратных скобках указывается индекс элемента).

ПОДРОБНОСТИ

Число `3` в инструкции `{0,3}` в команде `Console.Write("{0,3}", ((int[])objs[2])[i])` означает, что при отображении значения элемента числового массива выделяется не менее трех позиций.

Убедиться в том, что переменная `nums` и элемент `objs[2]` ссылаются на один и тот же массив, достаточно легко. Командой `((int[]) objs[2])[1]=0` меняем значение второго (по порядку) элемента в массиве, на который ссылается элемент `objs[2]`, а затем с помощью оператора цикла проверяем содержимое массива `nums`. Несложно заметить, что значение элемента `nums[1]` действительно изменилось.

Параметры командной строки

Фигуры, может, и нет, а характер — налицо.

из к/ф «Девчата»

При запуске программы на выполнение ей могут передаваться параметры, которые обычно называют *параметрами командной строки*. Если программа запускается из командной строки, то параметры указываются после имени файла программы через пробел. При работе со средой разработки параметры командной строки задаются на вкладке свойств проекта. Окно свойств проекта можно открывать по-разному. Например, в меню **Project** можно выбрать команду **Properties**, как показано на рис. 4.2.

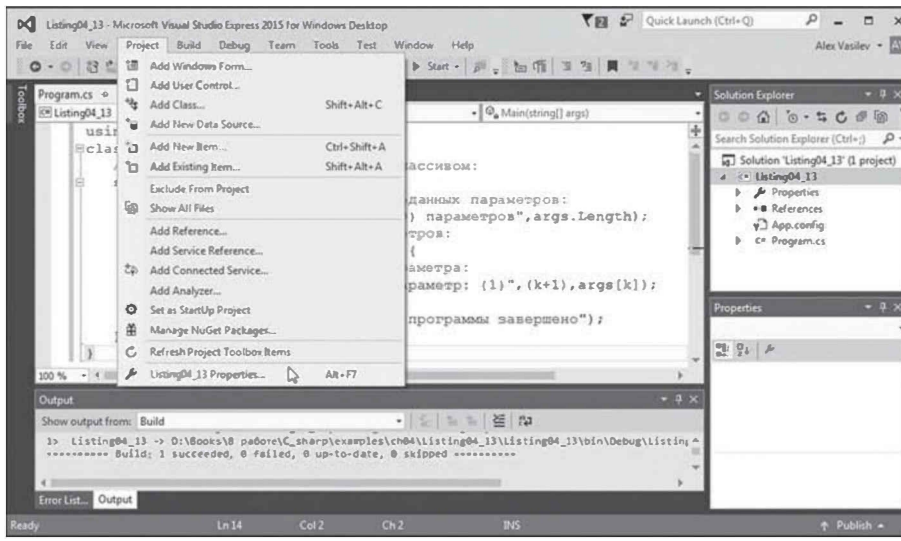


Рис. 4.2. Переход к окну свойств проекта (команда **Properties** в меню **Project**) для определения параметров командной строки

В результате откроется вкладка свойств проекта. В этой вкладке следует выбрать раздел **Debug**, как показано на рис. 4.3.

В данном разделе есть поле **Command line arguments**. Туда через пробел вводятся параметры командной строки. В данном примере мы вводим туда такие параметры:

```
первый 123 A 3,141592 последний
```

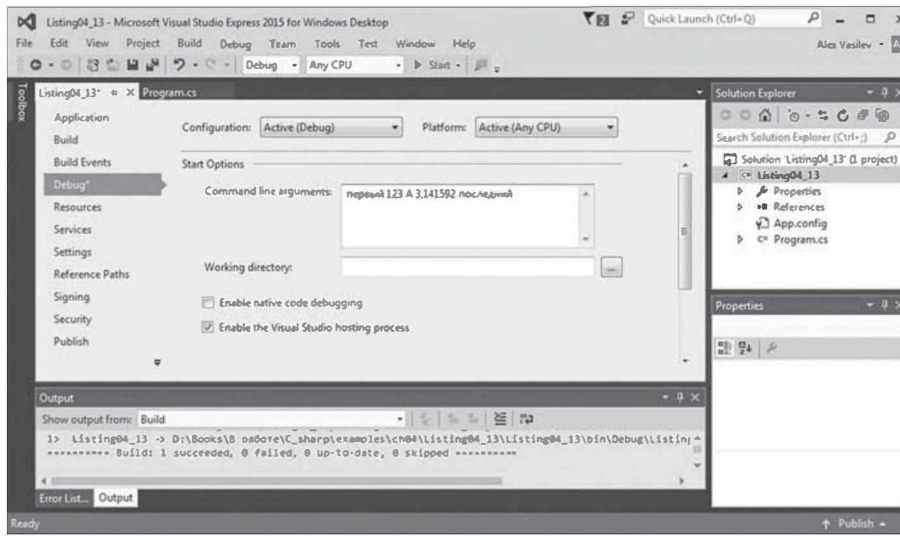


Рис. 4.3. Параметры командной строки определяются на вкладке свойств проекта в поле **Command line arguments** в разделе **Debug**

Это параметры, которые мы собираемся передавать программе при запуске. Но остается открытым вопрос о том, как эти параметры в программе можно использовать. Ответ простой: главный метод программы необходимо описать с аргументом, который является одномерным текстовым массивом. Параметры, которые передаются программе при запуске, отождествляются с элементами этого массива. Два важных момента:

- Параметры командной строки, какими бы они ни были по факту, передаются в текстовом формате (как элементы текстового массива). Если необходимо получить параметр в его «родном» формате, то выполняется явное преобразование из текстового формата к нужному типу.
- Текстовый массив, через который передаются параметры командной строки, создавать не нужно. В круглых скобках после названия главного метода программы объявляется переменная текстового массива. Эта переменная обрабатывается в программе как массив, и она отождествляется с массивом, сформированным параметрами командной строки.

Пример, в котором программе при запуске передаются параметры командной строки и эти параметры в программе обрабатываются, приведен в листинге 4.13.

Листинг 4.13. Параметры командной строки

```
using System;
class ProgArgsDemo{
   // Главный метод с аргументом – массивом:
   static void Main(string[] args){
      // Определение количества переданных параметров:
      Console.WriteLine("Передано {0} параметров", args.Length);
      // Отображение значений параметров:
      for(int k=0; k<args.Length; k++){
         // Отображение значения параметра:
         Console.WriteLine("{0}-й параметр: {1}",(k+1), args[k]);
      }
      Console.WriteLine("Выполнение программы завершено");
   }
}
```

Если программе передаются те параметры, что были указаны ранее, то в результате выполнения программы в консольном окне появляются такие сообщения.

Результат выполнения программы (из листинга 4.13)

```
Передано 5 параметров
1-й параметр: первый
2-й параметр: 123
3-й параметр: A
4-й параметр: 3,141592
5-й параметр: последний
Выполнение программы завершено
```

Первая особенность программы состоит в том, что метод `Main()` описан с аргументом `string[] args`. Поскольку массив формируется автоматически на основе переданных программе параметров, то количество элементов в массиве совпадает с количеством параметров. Следовательно, количество переданных параметров можно определить с помощью инструкции `args.Length` (размер массива `args`). Получение

текстового представления параметров — обращение к элементам массива `args` в формате `args[k]`. Сама программа очень простая: она выводит сообщение о количестве переданных программе параметров, а затем последовательно в консольном окне отображаются значения этих параметров.

Резюме

Как-то даже вот тянет устроить скандал.

из к/ф «Иван Васильевич меняет профессию»

- Массив представляет собой набор переменных, объединенных общим именем. Переменные, входящие в массив, называются элементами массива. Для идентификации элементов используется имя массива и индекс (или индексы). Количество индексов, необходимых для однозначной идентификации элемента массива, определяет размерность массива. Индексация всегда начинается с нуля. Размерность массива можно определить с помощью свойства `Rank`. Свойство `Length` позволяет определить количество элементов в массиве. Для определения размера массива по отдельным индексам используют метод `GetLength()` (аргумент метода определяет индекс, для которого вычисляется размер массива).
- Для реализации массива нужна переменная массива и собственно сам массив. Переменная массива содержит ссылку на массив. Для объявления переменной массива указываются идентификатор типа элементов массива, затем квадратные скобки и имя переменной. Для одномерного массива квадратные скобки пустые. Для многомерных массивов внутри квадратных скобок указывают запятые — количество запятых на единицу меньше размерности массива.
- Массив создается с помощью оператора `new`, после которого следуют идентификатор типа, определяющий тип элементов массива, и квадратные скобки. В квадратных скобках через запятую указывается размер массива по каждому из индексов. В результате выполнения инструкции на основе оператора `new` создается массив, а ссылка на этот массив является значением инструкции. Ссылку на массив можно записать в переменную массива (при условии, что тип и размерность массива совпадают с типом и размерностью, указанными при объявлении переменной массива).

- Массив при создании можно инициализировать. Для этого переменной массива значением присваивается список, содержащий значения, которыми инициализируется массив. Для одномерного массива список инициализации представляет собой последовательность разделенных запятыми значений, заключенных в фигурные скобки. Для двумерного массива элементами списка инициализации являются вложенные списки, которые содержат значения элементов в строках. Для трехмерного массива список инициализации содержит списки, элементами которых являются списки со значениями элементов трехмерного массива, и так далее.
- Для работы с массивами может использоваться оператор цикла `foreach`. В нем объявляется локальная переменная (того же типа, что и тип элементов массива). Эта локальная переменная при выполнении оператора цикла последовательно принимает значения элементов массива. Через локальную переменную можно прочитать значение элемента массива, но нельзя присвоить элементу новое значение.
- Можно создать массив, элементами которого являются переменные массива. В таком случае можно создать иллюзию двумерного массива со строками разной длины. Если создать массив объектных переменных класса `Object`, то элементам такого массива можно присваивать практически любые значения.
- При запуске программе можно передавать параметры (параметры командной строки). В программе такие параметры обрабатываются с помощью текстового массива, который объявляется как аргумент главного метода программы.

Задания для самостоятельной работы

— Я на следующий Новый год обязательно пойду в баню.
— Зачем же ждать целый год?

из к/ф «Ирония судьбы, или С легким паром»

1. Напишите программу, в которой создается одномерный числовой массив и заполняется числами, которые при делении на `5` дают в остатке `2` (числа `2`, `7`, `12`, `17` и так далее). Размер массива вводится пользователем. Предусмотреть обработку ошибки, связанной с вводом некорректного значения.

2. Напишите программу, в которой создается массив из 11 целочисленных элементов. Массив заполняется степенями двойки — числами $2^0 = 1, 2^1 = 2, 2^2 = 4, 2^3 = 8$ и так далее до $2^{10} = 1024$. При заполнении массива учесть, что начальный элемент равен 1, а каждый следующий больше предыдущего в 2 раза. Отобразить массив в консольном окне в прямом и обратном порядке. Размер массива задается переменной.

3. Напишите программу, в которой создается одномерный символьный массив из 10 элементов. Массив заполняется буквами «через одну», начиная с буквы `'a'`: то есть массив заполняется буквами `'a'`, `'c'`, `'e'`, `'g'` и так далее. Отобразите массив в консольном окне в прямом и обратном порядке. Размер массива задается переменной.

4. Напишите программу, в которой создается символьный массив из 10 элементов. Массив заполнить большими (прописными) буквами английского алфавита. Буквы берутся подряд, но только согласные (то есть гласные буквы `'A'`,`'E'` и `'I'` при присваивании значений элементам массива нужно пропустить). Отобразите содержимое созданного массива в консольном окне.

5. Напишите программу, в которой создается массив и заполняется случайными числами. Массив отображается в консольном окне. В этом массиве необходимо определить элемент с минимальным значением. В частности, программа должна вывести значение элемента с минимальным значением и индекс этого элемента. Если элементов с минимальным значением несколько, должны быть выведены индексы всех этих элементов.

6. Напишите программу, в которой создается целочисленный массив, заполняется случайными числами и после этого значения элементов в массиве сортируются в порядке убывания значений.

7. Напишите программу, в которой создается символьный массив, а затем порядок элементов в массиве меняется на обратный.

8. Напишите программу, в которой создается двумерный целочисленный массив. Он заполняется случайными числами. Затем в этом массиве строки и столбцы меняются местами: первая строка становится первым столбцом, вторая строка становится вторым столбцом и так далее. Например, если исходный массив состоял из 3 строк и 5 столбцов, то в итоге получаем массив из 5 строк и 3 столбцов.

9. Напишите программу, в которой создается и инициализируется двумерный числовой массив. Затем из этого массива удаляется строка и столбец (создается новый массив, в котором по сравнению с исходным удалена одна строка и один столбец). Индекс удаляемой строки и индекс удаляемого столбца определяется с помощью генератора случайных чисел.

10. Напишите программу, в которой создается двумерный числовой массив и этот массив заполняется «змейкой»: сначала первая строка (слева направо), затем последний столбец (сверху вниз), последняя строка (справа налево), первый столбец (снизу вверх), вторая строка (слева направо) и так далее.

Глава 5
СТАТИЧЕСКИЕ МЕТОДЫ

> Не надо меня щадить: нусть самое страшное, но нравда.
>
> *из к/ф «Бриллиантовая рука»*

Пришло время обсудить статические методы. В принципе, это тема, непосредственно относящаяся к классам и объектам. Поэтому мы, приступая к ее рассмотрению, немного забегаем вперед. Вместе с тем, такой подход представляется оправданным по нескольким причинам. Во-первых, впоследствии легче будет понять, как функционируют методы (не только статические, но и обычные) при их использовании с классами и объектами. Во-вторых, есть некоторые технологии, которые нам понадобятся в дальнейшем при анализе программных кодов. Знакомство, хоть и «преждевременное», со статическими методами облегчит нам эту задачу. В-третьих, мы обсудим статические методы лишь в той части, которая не подразумевает детального знакомства с принципами описания классов и создания объектов.

Среди вопросов, поднимаемых в этой главе, можно выделить следующие:

- особенности статических методов;
- принципы описания статических методов;
- перегрузка статических методов;
- передача массива аргументом статическому методу;
- методы, возвращающие результатом массив;
- механизмы передачи аргументов методам;
- использование рекурсии;
- методы с произвольным количеством аргументов.

Многие приемы и подходы, рассмотренные в этой главе, затем будут перенесены и на более общие случаи.

Знакомство со статическими методами

Вы убедили меня. Я понял, что вам еще нужен.

из к/ф «Старики-разбойники»

Метод представляет собой именованный блок команд, которые можно выполнять, вызывая метод. В этой главе будут обсуждаться статические методы, описанные в том же классе, в котором описан главный метод программы.

НА ЗАМЕТКУ

Метод в любом случае описывается в классе. Бывают методы статические и нестатические. Нестатический метод вызывается из объекта. То есть для вызова нестатического метода нужен объект: указывается имя объекта и через точку имя метода, который вызывается. Если метод статический, то для его вызова объект не нужен. При вызове статического метода вместо имени объекта указывается имя класса. Но если статический метод вызывается в том же классе, в котором он описан, то имя класса можно не указывать. Мы планируем вызывать статические методы в главном методе программы. Поэтому, если такие статические методы описаны в том же классе, в котором описан главный метод программы, то нет необходимости указывать имя класса. Именно такие ситуации рассматриваются в этой главе.

Для работы со статическими методами нам необходимо выяснить, как метод описывается и как метод вызывается. Начнем с описания метода.

Мы исходим из того, что метод описывается в том же классе, в котором содержится метод `Main()`, и вызывается описанный статический метод тоже в методе `Main()` (или в другом статическом методе, описанном в том же самом классе). Описание статического метода можно размещать как пред описанием метода `Main()`, так и после него. Ниже приведен шаблон описания статического метода:

```
static тип имя(аргументы){
   // Команды
}
```

Начинается описание статического метода с ключевого слова `static`. Ключевое слово «сигнализирует» о том, что метод статический. Затем в описании метода указывается идентификатор, определяющий тип

результата метода. Например, если метод результатом возвращает целочисленное значение, то в качестве идентификатора типа указываем ключевое слово `int`. Если метод результатом возвращает символ, то идентификатором типа результата метода указываем ключевое слово `char`. Может так быть, что метод не возвращает результат. В этом случае идентификатором типа результата указывается ключевое слово `void`.

ПОДРОБНОСТИ

Метод — это набор команд. Команды выполняются каждый раз, когда вызывается метод. Но кроме выполнения команд, метод может еще и возвращать значение. Возвращаемое методом значение — это значение инструкции, которой вызывается метод. Если метод возвращает значение, то команду вызова метода можно отождествлять с возвращаемым значением.

После ключевого слова, определяющего тип результата, следует имя метода — идентификатор, который выбирается пользователем. Он по возможности должен быть информативным и не может совпадать с зарезервированными ключевыми словами языка C#. После имени метода в круглых скобках описываются его аргументы. Дело в том, что при вызове методу могут передаваться некоторые значения, используемые методом в процессе выполнения команд и вычисления результата. Аргументы описываются так же, как объявляются переменные: указывается тип аргумента и его название. Аргументы метода разделяются запятыми. Тип указывается для каждого метода индивидуально — даже если они относятся к одному типу. Наконец, в фигурных скобках размещаются команды, выполняемые при вызове метода. В этих командах могут использоваться аргументы метода. Там также могут объявляться переменные, которые будем называть *локальными*.

НА ЗАМЕТКУ

Если метод описан в классе, то это совсем не означает, что команды в теле метода будут выполнены. Для выполнения команд в теле метода этот метод необходимо вызвать. Команды выполняются только тогда, когда вызывается метод. Если метод не вызывается, то и команды не выполняются. Переменные, которые объявлены в методе, доступны только в теле метода. Здесь уместно вспомнить главное правило, связанное с областью доступности переменных: переменная доступна в том блоке, в котором она объявлена. Если переменная объявлена в теле метода, то в теле метода она и доступна.

Поэтому такая переменная называется локальной. Место в памяти для локальной переменной выделяется при выполнении метода. А после того как метод завершает выполнение, переменная из памяти удаляется. При новом вызове метода все повторится: под переменную будет выделено место в памяти, а затем при завершении работы метода выделенное под переменную место освобождается.

Для того чтобы команды, описанные в методе, выполнились, метод необходимо вызвать. Вызывается метод просто: в соответствующем месте программного кода указываются имя метода и, если необходимо, аргументы, которые передаются методу при вызове. Аргументы указываются в круглых скобках после имени метода. Если у метода аргументов нет, то при вызове метода после имени метода указываются пустые круглые скобки.

НА ЗАМЕТКУ

При вызове метода аргументы ему передаются строго в том порядке, как они объявлены в описании метода.

Если метод вызывается с аргументами, то в результате вызова метода выполняются команды, описанные в теле метода, а вместо переменных, объявленных в описании метода в качестве аргументов, подставляются значения, фактически переданные методу при вызове.

Если метод не возвращает результат, то вызов метода сводится к выполнению команд из тела метода. Если метод возвращает результат, то в выражение, содержащее инструкцию вызова метода, вместо этой инструкции подставляется значение, возвращаемое методом.

При работе с методами часто используется инструкция `return`. Выполнение этой инструкции в теле метода завершает выполнение метода. Если после инструкции `return` указано некоторое выражение, то выполнение метода завершается, а значение выражения возвращается результатом метода. При этом тип значения выражения, указанного после `return`-инструкции, должен совпадать с идентификатором типа, указанным в описании метода как тип возвращаемого методом результата.

НА ЗАМЕТКУ

Если метод возвращает результат, то в теле метода используется `return`-инструкция (с указанием возвращаемого значения). Если метод не возвращает результат, то инструкцию `return` можно

не использовать (но если она используется, то после ключевого слова `return` ничего не указывается).

В листинге 5.1 представлена программа, в которой описываются и используются достаточно несложные статические методы.

Листинг 5.1. Знакомство со статическими методами

```
using System;
class StatMethDemo{
   // Статический метод для отображения текста,
   // переданного аргументом методу:
   static void show(string txt){
      Console.WriteLine(txt);
   }
   // Статический метод для вычисления факториала числа,
   // переданного аргументом методу:
   static int factorial(int n){
      // Локальная переменная:
      int s=1;
      // Вычисление произведения:
      for(int k=1; k<=n; k++){
         // Умножение произведения на число:
         s*=k;
      }
      // Результат метода:
      return s;
   }
   // Статический метод для возведения числа в степень.
   // Число и степень передаются аргументами методу:
   static double power(double x, int n){
      // Локальная переменная:
      double s=1;
      // Вычисление результата (число в степени):
```

```
        for(int k=1; k<=n; k++){
            // Текущее значение умножается на число:
            s*=x;
        }
        // Результат метода:
        return s;
    }
    // Главный метод программы:
    static void Main(){
        // Вызываем статический метод для отображения
        // сообщения в консольном окне:
        show("Начинаем вычисления:");
        int m=5;          // Целочисленные переменные
        double z=3, num; // Действительные переменные
        // Вычисление факториала числа:
        show(m+"!="+factorial(m));
        // Число в степени:
        num=power(z, m);
        // Отображение сообщения вызовом статического метода:
        show(z+" в степени "+m+": "+num);
    }
}
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 5.1)

```
Начинаем вычисления:
5!=120
3 в степени 5: 243
```

В этой программе, кроме главного метода, описывается еще и три статических метода. Эти статические методы затем вызываются в методе `Main()`. Проанализируем их программные коды.

Статический метод `show()` не возвращает результат (ключевое слово `void` перед названием метода в его описании), и у него один текстовый аргумент (описан как `string txt`). В теле метода всего одна команда `Console.WriteLine(txt)`, которой значение текстового аргумента метода отображается в консольном окне. Поэтому при вызове метода в консольном окне будет отображаться значение (текст), переданное методу аргументом.

Статический метод `factorial()` предназначен для вычисления факториала числа, переданного аргументом методу. Поэтому аргумент `n` метода `factorial()` объявлен как целочисленный (тип `int`). Вычисленное значение возвращается результатом метода. Тип результата определяется как целочисленный (тип `int`).

НА ЗАМЕТКУ

Факториал (обозначается как $n!$) числа n — это произведение чисел от 1 до данного числа включительно. Другими словами, по определению $n! = 1 \times 2 \times 3 \times \ldots \times n$. Например, $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$.

В теле метода командой объявляется целочисленная локальная переменная `s` с начальным значением `1`. После этого запускается оператор цикла, в котором индексная переменная `k` пробегает значения от `1` до `n` включительно (напомним, что `n` — это аргумент метода). За каждый цикл командой `s*=k` текущее значение переменной `s` умножается на значение индексной переменной `k`. В результате после завершения оператора цикла значение переменной `s` равно произведению натуральных чисел от `1` до `n`. Это именно то значение, которое нам нужно вычислить. Командой `return s` оно возвращается как результат метода.

ПОДРОБНОСТИ

В теле метода `factorial()` объявлена переменная `s`. Также в операторе цикла (в теле метода) объявлена переменная `k`. У этих переменных разная область доступности и разное время существования. Под переменную `s` место в памяти выделяется при вызове метода. Переменная (после объявления) доступна везде в теле метода. Существует переменная, пока выполняется метод.

Переменная `k` доступна только в операторе цикла. Под нее память выделяется при запуске на выполнение оператора цикла. После того как оператор цикла завершает работу, переменная `k` удаляется

из памяти (освобождается место в памяти, выделенное под эту переменную).

Еще один статический метод, описанный в программе, предназначен для вычисления результата возведения числа в целочисленную степень. Метод называется `power()`, у него два аргумента (первый типа `double` и второй типа `int`). Результатом метод возвращает значение типа `double`.

НА ЗАМЕТКУ

Речь идет о вычислении значения выражения вида x^n. По определению $x^n = x \times x \times \ldots \times x$ (всего n множителей). Значения x и n передаются аргументом методу `power()`, а результатом возвращается значение x^n.

Код метода `power()` немного напоминает код метода `factorial()`. В теле метода `power()` объявляется переменная `s` типа `double` с начальным значением `1`. Затем с помощью оператора цикла значение переменной `s` последовательно умножается `n` раз (второй аргумент метода) на значение `x` (первый аргумент метода). Значение переменной `s` возвращается результатом метода.

В главном методе программы сначала командой `show("Начинаем вычисления:")` вызывается статический метод `show()` для отображения текстового сообщения. Впоследствии этот же метод вызывается командой `show(m+"!="+factorial(m))`. В аргументе, переданном методу `show()`, использована инструкция `factorial(m)`, которой вызывается статический метод `factorial()`. Команда `show(m+"!="+factorial(m))` обрабатывается в несколько этапов. Сначала вычисляется текстовое значение `m+"!="+factorial(m)`, которое передается аргументом методу `show()` и, как следствие, отображается в консольном окне. Значение выражения `m+"!="+factorial(m)` вычисляется путем преобразования значения переменной `m` к текстовому формату, после чего к полученному тексту дописывается текст `"!="`. Затем вычисляется значение выражения `factorial(m)`, и оно дописывается к полученному ранее текстовому значению. Значение выражения `factorial(m)` — это факториал числа, определяемого значением переменной `m`.

ПОДРОБНОСТИ

Вычисление выражения `factorial(m)` выполняется так. Поскольку метод `factorial()` возвращает целочисленный результат, то при вызове метода для запоминания этого результата автоматически выделяется место в памяти. В процессе выполнения кода метода `factorial()` формируется значение переменной `s`. При выполнении команды `return s` в теле метода значение переменной `s` копируется в ячейку памяти, выделенную для хранения результата метода. Все локальные переменные, созданные при выполнении метода, удаляются, а значение из ячейки, выделенной под результат метода, используется в вычислениях — в данном случае подставляется вместо выражения `factorial(m)`.

Командой `num=power(z, m)` переменной `num` в качестве значения присваивается результат вызова метода `power()` с аргументами `z` и `m`.

ПОДРОБНОСТИ

При вызове метода `power()` с аргументами `z` и `m` значение `z` возводится в степень `m`, а результат записывается в локальную переменную `s`. При выполнении команды `return s` значение переменной `s` копируется в ячейку, автоматически выделенную для результата метода. При выполнении команды `num=power(z, m)` значение из ячейки с результатом метода копируется в переменную `num`. После этого ячейка с результатом метода удаляется из памяти.

Вычисленное значение переменной `num` используется в команде `show(z+" в степени "+m+": "+num)`.

Перегрузка статических методов

> Я вся такая внезапная, такая противоречивая вся...
>
> *из к/ф «Покровские ворота»*

В языке C# есть очень важный и полезный механизм, который называется *перегрузкой методов*. В частности, перегрузка может применяться и к статическим методам. Идея перегрузки состоит в том, что мы можем описать несколько методов с одним и тем же названием. В таком случае

обычно говорят про разные версии одного и того же метода, хотя в действительности методы разные. И они должны отличаться. Поскольку именем они не отличаются, то отличаться они должны списком аргументов (количеством и/или типом аргументов). При вызове перегруженного метода (то есть метода, у которого есть несколько версий) версия, которая будет вызвана, определяется по количеству и типу аргументов, фактически переданных методу. Поэтому при перегрузке метода его разные версии должны быть описаны так, чтобы отсутствовала любая неоднозначность при вызове метода. В противном случае программа просто не скомпилируется.

Пример, в котором используется перегрузка статических методов, представлен в листинге 5.2.

Листинг 5.2. Перегрузка статических методов

```
using System;
class OverloadMethDemo{
   // Версия статического метода для отображения текста
   // (с одним текстовым аргументом):
   static void show(string txt){
      Console.WriteLine("Текст: "+txt);
   }
   // Версия статического метода для отображения
   // целого числа (аргумент метода):
   static void show(int num){
      Console.WriteLine("Целое число: "+num);
   }
   // Версия статического метода для отображения
   // действительного числа (аргумент метода):
   static void show(double num){
      Console.WriteLine("Действительное число: "+num);
   }
   // Версия статического метода для отображения символа
   // (аргумент метода):
   static void show(char s){
```

```
      Console.WriteLine("Символ: "+s);
   }
   // Версия статического метода для отображения числа
   // (первый аргумент) и символа (второй аргумент):
   static void show(int num, char s){
      Console.WriteLine("Аргументы {0} и {1}", num, s);
   }
   // Главный метод программы:
   static void Main(){
      // Целочисленная переменная:
      int num=5;
      // Действительная числовая переменная:
      double z=12.5;
      // Символьная переменная:
      char symb='W';
      // Вызываем метод с символьным аргументом:
      show(symb);
      // Вызываем метод с текстовым аргументом:
      show("Знакомимся с перегрузкой методов");
      // Вызываем метод с целочисленным аргументом:
      show(num);
      // Вызываем метод с действительным аргументом:
      show(z);
      // Вызываем метод с двумя аргументами:
      show(num,'Q');
   }
}
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 5.2)

```
Символ: W
Текст: Знакомимся с перегрузкой методов
```

```
Целое число: 5
Действительное число: 12,5
Аргументы 5 и Q
```

В представленной программе описано пять версий статического метода `show()`:

- в программе есть версия метода с текстовым аргументом;
- описана версия метода с целочисленным аргументом;
- есть версия метода с действительным числовым аргументом;
- описана версия метода с символьным аргументом;
- а также версия метода с двумя аргументами (целочисленным и символьным).

Во всех случаях метод не возвращает результат. Каждая из версий метода отображает в консольном окне значения переданного аргумента (или аргументов). Вопрос только в том, какой вспомогательный текст (помимо значения аргументов) появляется в консоли.

В главном методе программы есть несколько команд, которыми вызывается метод `show()`. Но аргументы методу передаются разные. При выполнении соответствующей команды на основании тех аргументов, которые переданы методу, выбирается версия метода для вызова. Например, при выполнении команды `show(symb)` вызывается версия метода `show()` с одним символьным аргументом, поскольку переменная `symb`, переданная аргументом методу при вызове, объявлена как символьная. А вот при выполнении команды `show(num,'Q')` вызывается версия метода `show()` с первым целочисленным аргументом и вторым символьным аргументом.

ПОДРОБНОСТИ

Для выбора версии метода должно совпадать не только количество аргументов, но еще и тип каждого аргумента. Допустим, в программе вместо версии метода `show()` с целочисленным и символьным аргументами описана версия метода `show()` с двумя символьными аргументами. Тогда команда `show(num,'Q')` была бы некорректной, поскольку в программе не оказалось бы подходящей версии вызываемого метода. Такие программные коды не компилируются. С другой стороны, если мы опишем вместо нынешней версии метода `show()` с целочисленным и символьным аргументами версию

метода show() с двумя целочисленными аргументами, то программа скомпилируется и команда show(num,'Q') выполнится так: вызывается метод show() с двумя целочисленными аргументами, и вместо символа 'Q' используется его код 81 в кодовой таблице. То есть здесь выполняется автоматическое приведение значения типа char в значение типа int. А вот обратное преобразование из типа int в тип char автоматически не выполняется, поэтому вариант метода show() с двумя символьными аргументами не сработает.

Аналогично, если мы уберем из программы описание метода show() с целочисленным аргументом, то при выполнении команды show(num) будет вызвана версия метода с double-аргументом и целочисленное значение переменной num автоматически преобразуется в значение типа double.

Массив как аргумент метода

У папы там совсем не то, что он всем говорит.

из к/ф «Бриллиантовая рука»

Аргументами методам можно передавать практически все, что угодно. Среди этого «всего» есть и массивы. Другими словами, аргументом методу можно передать массив (одномерный, двумерный, многомерный). Чтобы понять, как это делается, следует вспомнить, как мы получаем доступ к массиву. Используется переменная массива — то есть переменная, которая ссылается на массив и которую в большинстве случаев отождествляют с массивом. Поэтому, если у нас имеется массив, который следует передать аргументом методу, то сделать это можно, передав методу ссылку на массив. Этого будет вполне достаточно для того, чтобы метод получил доступ к массиву. Именно такой подход используется в C#.

С технической точки зрения, при передаче массива методу соответствующий аргумент описывается точно так же, как объявляется переменная массива соответствующего типа. Как данный подход используется на практике, проиллюстрировано в программе в листинге 5.3.

Листинг 5.3. Передача массива аргументом методу

```
using System;
class ArrayToMethDemo{
```

```
// Метод для заполнения массива случайными числами:
static void fillRand(int[] nums){
   // Объект для генерирования случайных чисел:
   Random rnd=new Random();
   // Заполнение массива случайными числами:
   for(int k=0; k<nums.Length; k++){
      nums[k]=rnd.Next(1,101);
   }
}
// Метод для отображения одномерного
// целочисленного массива:
static void showArray(int[] nums){
   // Перебор элементов массива:
   for(int k=0; k<nums.Length; k++){
      // Отображение значения элемента:
      Console.Write("| {0}", nums[k]);
   }
   Console.WriteLine("|");
}
// Метод для отображения двумерного
// целочисленного массива:
static void showArray(int[,] nums){
   // Перебор строк в массиве:
   for(int i=0; i<nums.GetLength(0); i++){
      // Перебор элементов в строке:
      for(int j=0; j<nums.GetLength(1); j++){
         // Отображение значения элемента массива:
         Console.Write("{0,3}", nums[i, j]);
      }
      // Переход к новой строке:
      Console.WriteLine();
   }
```

```
}
// Метод для вычисления наименьшего элемента в массиве:
static int findMin(int[] nums){
   // Локальная переменная:
   int s=nums[0];
   // Поиск наименьшего значения:
   for(int k=1; k<nums.Length; k++){
      // Если проверяемый элемент имеет значение,
      // меньшее текущего значения переменной s:
      if(nums[k]<s) s=nums[k];
   }
   // Результат метода:
   return s;
}
// Главный метод программы:
static void Main(){
   // Одномерные массивы:
   int[] A={1,3,5,7,9,11,13,15};
   int[] B=new int[5];
   // Двумерный массив:
   int[,] C={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
   // Массив B заполняется случайными числами:
   fillRand(B);
   Console.WriteLine("Одномерный массив A:");
   // Отображается массив A:
   showArray(A);
   Console.WriteLine("Одномерный массив B:");
   // Отображается массив B:
   showArray(B);
   // Поиск наименьшего элемента:
   int m=findMin(B);
   Console.WriteLine("Наименьшее значение: {0}", m);
```

```
        Console.WriteLine("Двумерный массив C:");
        // Отображается массив C:
        showArray(C);
    }
}
```

Результат выполнения программы такой (учтите, что используется генератор случайных чисел).

Результат выполнения программы (из листинга 5.3)

```
Одномерный массив A:
| 1 | 3 | 5 | 7 | 9 | 11 | 13 | 15 |
Одномерный массив B:
| 43 | 7 | 90 | 23 | 52 |
Наименьшее значение: 7
Двумерный массив C:
  1  2  3  4
  5  6  7  8
  9 10 11 12
```

В программе описан метод `fillRand()`, предназначенный для заполнения одномерного целочисленного массива случайными числами. Метод не возвращает результат, а аргумент метода объявлен инструкцией `int[] nums`, которая формально совпадает с объявлением переменной массива. В теле метода командой `Random rnd=new Random()` создается объект `rnd` класса `Random`, который используется в теле метода для генерирования случайных чисел. Это происходит в операторе цикла, в котором индексная переменная `k` перебирает индексы элементов массива, объявленного аргументом метода. Значение элементу присваивается командой `nums[k]=rnd.Next(1,101)` (случайное целое число в диапазоне значений от `1` до `100` включительно).

Для отображения содержимого массивов описаны две версии метода `showArray()`. Одна версия предназначена для отображения содержимого одномерного массива, а другая версия метода предназначена для отображения содержимого двумерного целочисленного массива. Если аргументом методу `showArray()` передается одномерный массив

(объявлен как `int[] nums`), то элементы массива отображаются в одной строке с использованием вертикальной черты в качестве разделителя.

Если аргументом методу `showArray()` передается двумерный массив (аргумент объявлен как `int[,] nums`), то элементы массива отображаются в консольном окне построчно.

Еще один метод, который называется `findMin()`, получает аргументом одномерный целочисленный массив, а результатом возвращает целочисленное значение, которое совпадает со значением наименьшего элемента в массиве. В теле метода объявляется локальная переменная `s` с начальным значением `nums[0]` (значение первого элемента массива, переданного аргументом методу). После этого запускается оператор цикла, в котором перебираются все элемента массива, кроме начального, и если значение проверяемого элемента `nums[k]` меньше текущего значения переменной `s` (условие `nums[k]<s` в условном операторе в теле оператора цикла), то командой `s=nums[k]` переменной `s` присваивается новое значение. В итоге после завершения оператора цикла в переменную `s` будет записано значение наименьшего элемента в массиве. Командой `return s` значение переменной `s` возвращается результатом метода.

В главном методе программы командой `int[] A={1,3,5,7,9,11,13,15}` создается и инициализируется одномерный массив `A`. Командой `int[] B=new int[5]` просто создается одномерный массив `B` (но не заполняется). Для заполнения массива случайными числами использована команда `fillRand(B)`.

ПОДРОБНОСТИ

При создании целочисленных массивов они автоматически заполняются нулями. Вместе с тем, хороший стиль программирования состоит в том, чтобы заполнять массивы явным образом (даже если их нужно заполнить нулями).

Командой `int[,] C={{1,2,3,4},{5,6,7,8},{9,10,11,12}}` создается и инициализируется двумерный целочисленный массив из трех строк и четырех столбцов. Для отображения содержимого массивов используются команды `showArray(A)`, `showArray(B)` и `showArray(C)`. Также для массива `B` вычисляется наименьшее значение элемента. Для этой цели использован метод `findMin()`: командой `int m=findMin(B)` объявляется переменная `m`, значением которой присваивается результат вызова метода `findMin()` с аргументом `B`.

Массив как результат метода

Ну а это довесок к кошмару.

из к/ф «Старики-разбойники»

Метод может возвращать результатом массив. Точнее, метод может возвращать ссылку на массив. Обычно схема такая: при вызове метода создается массив, а результатом метод возвращает ссылку на него. Как уже отмечалось, переменные, которые создаются при вызове метода, существуют, пока метод выполняется, а после завершения метода они удаляются из памяти. В принципе, массив, который создан при вызове метода, также является «кандидатом на удаление». Но если ссылка на массив возвращается методом как результат и присваивается в качестве значения, например, какой-то переменной, то эта переменная (объявленная до вызова метода) будет ссылаться на массив. Такой массив не будет удален из памяти даже после завершения выполнения метода.

ПОДРОБНОСТИ

В языке C# существует система автоматического сбора мусора. Если в памяти есть объект или массив, на который не ссылается ни одна переменная в программе, такой объект/массив будет из памяти удален системой сборки мусора. Так, если при вызове метода создается массив и ссылка на него записывается в локальную переменную, но как результат этот массив (ссылка на него) не возвращается, то после выполнения метода локальная переменная из памяти удаляется. Следовательно, на созданный массив не будет ссылок, и он удаляется системой сборки мусора. Но если ссылка на массив возвращается результатом метода и записывается во внешнюю (для метода) переменную, то после завершения работы метода на созданный массив в программе будет ссылка. Поэтому массив не удаляется из памяти.

Если метод возвращает результатом ссылку на массив, то тип результата метода указывается точно так же, как он указывается для переменной массива того же типа. Например, если метод возвращает результатом ссылку на одномерный целочисленный массив, то идентификатор типа для такого метода выглядит как `int[]`. Если метод возвращает результатом ссылку на двумерный символьный массив, то идентификатор типа результата метода имеет вид `char[,]`.

В листинге 5.4 представлена программа, в которой описано несколько статических методов, возвращающих результатом массив.

Листинг 5.4. Массив как результат метода

```
using System;
class ArrayFromMethDemo{
   // Метод для создания массива с числами Фибоначчи:
   static int[] fibs(int n){
      // Создается массив:
      int[] nums=new int[n];
      // Первый элемент массива:
      nums[0]=1;
      // Если массив из одного элемента:
      if(nums.Length==1) return nums;
      // Второй элемент массива:
      nums[1]=1;
      // Заполнение элементов массива:
      for(int k=2; k<nums.Length; k++){
         // Значение элемента массива равно сумме значений
         // двух предыдущих элементов:
         nums[k]=nums[k-1]+nums[k-2];
      }
      // Результат метода – ссылка на массив:
      return nums;
   }
   // Метод для создания массива со случайными символами:
   static char[] rands(int n){
      // Объект для генерирования случайных чисел:
      Random rnd=new Random();
      // Создание массива:
      char[] symbs=new char[n];
      // Заполнение массива:
```

```
    for(int k=0; k<symbs.Length; k++){
        // Значение элемента – случайный символ:
        symbs[k]=(char)('A'+rnd.Next(26));
    }
    // Результат метода – ссылка на массив:
    return symbs;
}
// Метод для создания двумерного массива с нечетными
// числами:
static int[,] odds(int m, int n){
    // Создание двумерного массива:
    int[,] nums=new int[m, n];
    // Локальная переменная:
    int val=1;
    // Перебор строк массива:
    for(int i=0; i<nums.GetLength(0); i++){
        // Перебор элементов в строке:
        for(int j=0; j<nums.GetLength(1); j++){
            // Значение элемента:
            nums[i, j]=val;
            // Значение для следующего элемента:
            val+=2;
        }
    }
    // Результат метода – ссылка на массив:
    return nums;
}
// Главный метод программы:
static void Main(){
    // Переменная для целочисленного массива:
    int[] A;
    // Переменная для символьного массива:
```

```
        char[] B;
        // Переменная для двумерного массива:
        int[,] C;
        // Создается массив с числами Фибоначчи:
        A=fibs(10);
        Console.WriteLine("Числа Фибоначчи:");
        // Отображение содержимого массива:
        foreach(int s in A){
           Console.Write("| {0}", s);
        }
        Console.WriteLine("|");
        // Создается массив со случайными символами:
        B=rands(8);
        Console.WriteLine("Случайные символы:");
        // Отображение содержимого массива:
        foreach(char s in B){
           Console.Write("| {0}", s);
        }
        Console.WriteLine("|");
        // Создается двумерный массив с нечетными числами:
        C=odds(4,6);
        Console.WriteLine("Двумерный массив:");
        // Отображение содержимого двумерного массива:
        for(int i=0; i<C.GetLength(0); i++){
           for(int j=0; j<C.GetLength(1); j++){
              Console.Write("{0,4}", C[i, j]);
           }
           Console.WriteLine();
        }
     }
}
```

В результате выполнения программы в консольном окне могут появиться такие сообщения (нужно учесть, что в программе использован генератор случайных чисел).

Результат выполнения программы (из листинга 5.4)

```
Числа Фибоначчи:
| 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 |
Случайные символы:
| R | P | E | P | C | T | K | U |
Двумерный массив:
  1   3   5   7   9  11
 13  15  17  19  21  23
 25  27  29  31  33  35
 37  39  41  43  45  47
```

В программе описывается три статических метода. Каждый из них возвращает результатом ссылку на массив. Метод `fibs()` создает одномерный целочисленный массив, заполняет его числами Фибоначчи, а результатом возвращает ссылку на этот массив.

НА ЗАМЕТКУ

Последовательность Фибоначчи формируется так: первые два числа в последовательности равны 1, а каждое следующее число — это сумма двух предыдущих. То есть речь идет о последовательности чисел 1, 1, 2, 3, 5, 8, 13, 21 и так далее.

Аргументом методу передается целое число, определяющее размер массива. Идентификатором типа результата указана инструкция `int[]`. В теле метода командой `int[] nums=new int[n]` создается одномерный целочисленный массив. Размер массива определяется аргументом `n`, переданным методу. Командой `nums[0]=1` первому (начальному) элементу массива присваивается единичное значение. Затем выполняется условный оператор. В нем проверяется условие `nums.Length==1`, истинное в случае, если массив состоит всего из одного элемента. Если так, то получается, что массив уже заполнен (на предыдущем этапе единственному элементу массива было присвоено единичное значение). Поэтому командой `return nums` завершается выполнение метода,

а результатом возвращается ссылка на созданный массив (напомним, что выполнение инструкции `return` в теле метода приводит к завершению выполнения метода). Если же условие в условном операторе ложно, то командой `nums[1]=1` единичное значение присваивается и второму (по порядку) элементу массива.

НА ЗАМЕТКУ

Получается так, что нам нужно явно присвоить первым двум элементам массива единичные значения. Но у нас нет гарантии, что второй элемент в массиве существует. Поэтому после присваивания значения первому элементу выполняется проверка на предмет того, равен ли размер массива единице. Если это так, то с помощью `return`-инструкции выполнение метода завершается. Если выполнение метода не завершилось (размер массива не равен единице), то присваивается значение второму элементу.

После того как первым двум элементам присвоены значения, с помощью оператора цикла перебираются прочие элементы массива, и при заданном индексе `k` командой `nums[k]=nums[k-1]+nums[k-2]` значение элемента массива вычисляется как сумма значений двух предыдущих элементов. По завершении оператора цикла, когда массив заполнен, командой `return nums` ссылка на массив возвращается результатом метода.

ПОДРОБНОСТИ

Если окажется, что массив состоит всего из двух элементов, то при первой проверке условия `k<nums.Length` в операторе цикла с учетом того, что начальное значение переменной `k` равно `2`, условие окажется ложным. Поэтому команды в теле оператора цикла выполняться не будут.

Метод `rands()` предназначен для создания одномерного массива, заполненного случайными символами. Результат массива имеет тип `char[]` (ссылка на символьный массив), а аргументом методу передается целое число, определяющее размер массива. В теле метода создается объект `rnd` класса `Random` для генерирования случайных чисел (команда `Random rnd=new Random()`). Командой `char[] symbs=new char[n]` создается символьный массив. После этого запускается оператор цикла, в котором индексная переменная `k` перебирает значения

индексов элементов созданного массива. За каждый цикл командой `symbs[k]=(char)('A'+rnd.Next(26))` элементу массива присваивается случайное символьное значение. После заполнения массива командой `return symbs` ссылка на него возвращается результатом метода.

ПОДРОБНОСТИ

Значением выражения `rnd.Next(26)` является случайное целое число в диапазоне от `0` до `25` включительно. Выражение `'A'+rnd.Next(26)` вычисляется так: к коду `65` символа `'A'` прибавляется значение выражения `rnd.Next(26)`. Получается какое-то (случайное) целое число в диапазоне значений от `65` до `90`. После явного приведения к символьному типу (инструкция `(char)` перед выражением `'A'+rnd.Next(26)`) получаем символ с соответствующим кодом.

Метод `odds()` создает двумерный целочисленный массив, заполняет его нечетными числами и возвращает ссылку на этот массив. У метода два целочисленных аргумента, которые определяют количество строк и столбцов в двумерном массиве. Идентификатор типа результата имеет вид `int[,]` (ссылка на двумерный целочисленный массив).

В теле метода двумерный массив создается командой `int[,] nums=new int[m, n]`. Для заполнения массива использованы вложенные операторы цикла. При заданных индексах `i` и `j` значение элемента массива вычисляется командой `nums[i, j]=val`. После этого командой `val+=2` текущее значение переменной `val` увеличивается на `2`. Поэтому значение следующего элемента будет на `2` больше. Начальное значение переменной `val` равно `1`. В итоге элементы массива заполняются числами, начиная со значения `1`, и каждое следующее значение больше предыдущего на `2`. А это последовательность нечетных чисел. После заполнения массива ссылка на него возвращается результатом метода.

НА ЗАМЕТКУ

Желающие могут подумать, как следует изменить программный код, чтобы массив заполнялся четными числами или, например, числами, которые кратны 5 (то есть 5, 10, 15, 20 и так далее).

В главном методе программы объявляется несколько переменных массива: переменная `A` для одномерного целочисленного массива, переменная `B` для одномерного символьного массива и переменная `C` для двумерного

целочисленного массива. Сами массивы не создаются. Точнее, они создаются при вызове статических методов, описанных в программе. При выполнении команды `A=fibs(10)` создается массив из `10` (аргумент метода `fibs()`) чисел Фибоначчи, и ссылка на массив записывается в переменную `A`. Командой `B=rands(8)` создается массив из `8` (аргумент метода `rands()`) случайных символов, и ссылка на этот массив записывается в переменную `B`. Наконец, командой `C=odds(4,6)` создается двумерный массив из `4` строк и `6` столбцов (аргументы метода `odds()`), и ссылка на массив записывается в переменную `C`. Массив заполнен нечетными числами. Также в главном методе есть команды (операторы цикла), предназначенные для отображения содержимого созданных массивов. Читателю они уже должны быть понятны, поэтому особых комментариев не требуют.

НА ЗАМЕТКУ

Допустим, имеется два метода. Один метод, аргументом которому передается ссылка на целочисленный массив, при вызове заполняет этот массив случайными числами. Другой метод, аргументом которому передается целое число (размер создаваемого массива), возвращает ссылку на целочисленный массив, заполненный случайными числами. В чем между ними принципиальная разница? В первом случае мы сами создаем массив и ссылку на него передаем методу. Во втором случае массив создается при вызове метода.

Механизмы передачи аргументов методу

> Начинаю действовать без шума и ныли по вновь утвержденному плану.
>
> *из к/ф «Бриллиантовая рука»*

Есть два механизма (или способа) передачи аргументов в метод — по *значению* и по *ссылке*. По умолчанию в C# аргументы в метод передаются по значению. Это подразумевает, что в действительности в метод передается не та переменная, которую мы непосредственно указали аргументом метода при вызове, а ее техническая копия. Каждый раз, когда мы вызываем некоторый метод и передаем ему аргументы, автоматически для каждого из аргументов создается копия. Все операции при выполнении команд метода выполняются с этой копией. В большинстве случаев

не играет особой роли, как именно аргументы передаются в метод. Важным это становится, когда мы хотим при вызове метода изменить значения аргументов, переданных ему. Для большей конкретики рассмотрим пример, представленный в листинге 5.5.

Листинг 5.5. Механизм передачи аргументов методу

```
using System;
class ArgsDemo{
   // Первый метод. Аргумент – целое число:
   static void alpha(int n){
      // Проверка значения аргумента:
      Console.WriteLine("В методе alpha(). На входе: "+n);
      // Попытка изменить значение аргумента:
      n++;
      // Проверка значения аргумента:
      Console.WriteLine("В методе alpha(). На выходе: "+n);
   }
   // Второй метод. Аргумент – ссылка на массив:
   static void bravo(int[] n){
      // Проверка содержимого массива:
      Console.WriteLine("В методе bravo(). На входе: "+ArrayToText(n));
      // Перебор элементов массива:
      for(int k=0; k<n.Length; k++){
         // Попытка изменить значение элемента массива:
         n[k]++;
      }
      // Проверка содержимого массива:
      Console.WriteLine("В методе bravo(). На выходе: "+ArrayToText(n));
   }
   // Третий метод. Аргумент – ссылка на массив:
   static void charlie(int[] n){
      // Проверка содержимого массива:
      Console.WriteLine("В методе charlie(). На входе: "+ArrayToText(n));
```

```
        // Создается новый массив:
        int[] m=new int[n.Length];
        // Перебор элементов в массиве:
        for(int k=0; k<n.Length; k++){
            // Значение элемента массива:
            m[k]=n[k]+1;
        }
        // Попытка присвоить новое значение аргументу:
        n=m;
        // Проверка содержимого массива:
        Console.WriteLine("В методе charlie(). На выходе: "+ArrayToText(n));
    }
    // Метод для преобразования массива в текст:
    static string ArrayToText(int[] n){
        // Текстовая переменная:
        string res="["+n[0];
        // Перебор элементов массива (кроме начального):
        for(int k=1; k<n.Length; k++){
            // Дописывание текста в текстовую переменную:
            res+=","+n[k];
        }
        // Дописывание текста в текстовую переменную:
        res+="]";
        // Результат метода – текстовая строка:
        return res;
    }
    // Главный метод программы:
    static void Main(){
        // Переменная для передачи аргументом методу:
        int A=100;
        // Проверка значения переменной:
        Console.WriteLine("До вызова метода alpha(): A="+A);
```

```
        // Вызов метода:
        alpha(A);
        // Проверка значения переменной:
        Console.WriteLine("После вызова метода alpha: A="+A);
        // Массив для передачи аргументом методу:
        int[] B={1,3,5};
        // Проверка содержимого массива:
        Console.WriteLine("До вызова метода bravo(): B="+ArrayToText(B));
        // Вызов метода:
        bravo(B);
        // Проверка содержимого массива:
        Console.WriteLine("После вызова метода bravo(): B="+ArrayToText(B));
        // Массив для передачи аргументом методу:
        int[] C={2,4,6};
        // Проверка содержимого массива:
        Console.WriteLine("До вызова метода charlie(): C="+ArrayToText(C));
        // Вызов метода:
        charlie(C);
        // Проверка содержимого массива:
        Console.WriteLine("После вызова метода charlie(): C="+ArrayToText(C));
    }
}
```

В программе описаны три метода: `alpha()`, `bravo()` и `charlie()`. В каждом из этих методов выполняется попытка изменить аргумент (или создается иллюзия, что аргумент изменяется). Но аргументы у методов разные. У метода `alpha()` аргумент — целое число. У методов `bravo()` и `charlie()` аргумент — ссылка на одномерный целочисленный массив. Проанализируем коды этих методов. Начнем с метода `alpha()`.

При вызове метода появляется сообщение: в консольное окно выводится значение аргумента (обозначен как `n`), переданного методу. После этого командой `n++` выполняется попытка увеличить на единицу значение

аргумента. Затем снова в консольное окно выводится сообщение с уже «новым» значением аргумента.

При выполнении метода `bravo()` в консольном окне отображается содержимое массива (обозначен как `n`), переданного аргументом методу. Затем с помощью оператора цикла перебираются все элементы массива, и значение каждого элемента увеличивается на единицу (команда `n[k]++` в теле оператора цикла). После этого снова отображается содержимое массива.

ПОДРОБНОСТИ

При отображении содержимого массивов в программе использован вспомогательный метод `ArrayToText()`. Аргументом методу передается ссылка на целочисленный одномерный массив. Результатом метод возвращает текстовую строку, которая содержит значения элементов массива. Значения заключены в квадратные скобки и разделены запятыми. Поэтому для отображения содержимого массива достаточно передать массив аргументом методу `ArrayToText()` и вывести на экран результат, возвращаемый методом.

Методу `charlie()` так же, как и методу `bravo()`, передается ссылка на целочисленный массив. Но операции, выполняемые с аргументом и массивом, в данном случае несколько сложнее. В частности, при выполнении метода `charlie()` создается новый массив `m` (команда `int[] m=new int[n.Length]`). Размер этого массива точно такой же, как и размер массива `n`, переданного аргументом методу. Новый массив заполняется поэлементно. Для этого использован оператор цикла, в теле которого командой `m[k]=n[k]+1` значение элемента созданного массива вычисляется как соответствующее значение переданного аргументом массива, увеличенное на единицу. После завершения оператора цикла командой `n=m` выполняется попытка изменить значение аргумента. Если подойти к этой команде чисто формально, то мы пытаемся перебросить ссылку из аргумента `n` с исходного массива (переданного через ссылку аргументом методу) на массив, который был создан в методе (хотя в реальности все не так просто). После выполнения команды проверяется содержимое массива, на который ссылается переменная `n`.

В главном методе программы объявляется переменная `A` со значением `100` и объявляются и инициализируются два целочисленных массива `B` и `C`. Мы проверяем работу методов `alpha()`, `bravo()` и `charlie()`:

- Проверяется значение переменной A, потом эта переменная передается аргументом методу alpha(), и после вызова метода снова проверяется значение переменной A.
- Проверяется содержимое массива B, массив передается аргументом методу bravo(), и снова проверяется содержимое массива B.
- Проверяется содержимое массива C, массив передается аргументом методу charlie(), после чего снова проверяется содержимое массива C.

Результат выполнения программы такой (места, на которые следует обратить внимание, выделены жирным шрифтом).

Результат выполнения программы (из листинга 5.5)

```
До вызова метода alpha(): A=100
В методе alpha(). На входе: 100
В методе alpha(). На выходе: 101
После вызова метода alpha: A=100
До вызова метода bravo(): B=[1,3,5]
В методе bravo(). На входе: [1,3,5]
В методе bravo(). На выходе: [2,4,6]
После вызова метода bravo(): B=[2,4,6]
До вызова метода charlie(): C=[2,4,6]
В методе charlie(). На входе: [2,4,6]
В методе charlie(). На выходе: [3,5,7]
После вызова метода charlie(): C=[2,4,6]
```

Что мы видим? Значение 100 переменной A после вызова метода alpha() не изменилось, хотя при второй проверке аргумента в методе alpha() значение было 101. Чтобы понять этот «парадокс», следует учесть, что аргументы по умолчанию передаются по значению. Поэтому при выполнении команды alpha(A) вместо переменной A в метод alpha() передается техническая копия этой переменной. Ее значение — такое же, как у переменной A. Поэтому при проверке значения аргумента появляется значение 100. При попытке увеличить значение аргумента метода на 1 в действительности увеличивается значение технической переменной. Ее же значение проверяется в методе, и получаем число 101. Когда метод alpha() завершает работу, техническая

переменная удаляется из памяти. А переменная A остается со своим исходным значением 100.

Что касается массива B, то он после выполнения команды bravo(B) изменился. Причина в том, что здесь на самом деле мы не пытаемся изменить аргумент метода. Им является ссылка на массив, а изменяем мы массив. Более детально все происходит следующим образом. При передаче переменной B аргументом методу bravo() для нее создается техническая копия. Значение у копии переменной B такое же, как и у самой переменной B. И это адрес массива. Поэтому и переменная B, и ее копия ссылаются на один и тот же массив. Когда при выполнении метода bravo() изменяются значения элементов массива, то, хотя они изменяются через копию переменной B, речь идет о том же массиве, на который ссылается переменная B. Отсюда и результат.

Иная ситуация при вызове метода chalrie() с аргументом C. Массив, на который ссылается переменная C, после вызова метода не изменяется. Причина в том, что в теле метода charlie() мы пытаемся записать в переменную, переданную аргументом, новую ссылку. Но поскольку, как и во всех предыдущих случаях, аргументом передается не сама переменная, а ее копия, то новое значение присваивается копии переменной C. Копия ссылается на новый созданный массив. Когда мы проверяем содержимое массива (после присваивания n=m) в теле метода, то проверяется новый созданный массив. Массив, на который ссылается переменная C, не меняется.

Кроме используемого по умолчанию режима передачи аргументов по значению, аргументы еще можно передавать и по *ссылке*. В таком случае в метод передается именно та переменная, которая указана аргументом (а не ее копия). Если мы хотим, чтобы аргументы в метод передавались по ссылке, то в описании метода перед соответствующим аргументом указывается инструкция ref. Такая же инструкция указывается перед аргументом при вызове метода. Как иллюстрацию рассмотрим программу в листинге 5.6. Это модификация предыдущего примера (см. листинг 5.5), но только аргументы методам alpha(), bravo() и charlie() передаются по ссылке. Для удобства и сокращения объема кода все комментарии удалены, а места появления инструкции ref выделены жирным шрифтом.

Листинг 5.6. Передача аргументов по ссылке

```
using System;
class RefArgsDemo{
   static void alpha(ref int n){
```

```
    Console.WriteLine("В методе alpha(). На входе: "+n);
    n++;
    Console.WriteLine("В методе alpha(). На выходе: "+n);
}
static void bravo(ref int[] n){
    Console.WriteLine("В методе bravo(). На входе: "+ArrayToText(n));
    for(int k=0; k<n.Length; k++){
        n[k]++;
    }
    Console.WriteLine("В методе bravo(). На выходе: "+ArrayToText(n));
}
static void charlie(ref int[] n){
    Console.WriteLine("В методе charlie(). На входе: "+ArrayToText(n));
    int[] m=new int[n.Length];
    for(int k=0; k<n.Length; k++){
        m[k]=n[k]+1;
    }
    n=m;
    Console.WriteLine("В методе charlie(). На выходе: "+ArrayToText(n));
}
static string ArrayToText(int[] n){
    string res="["+n[0];
    for(int k=1; k<n.Length; k++){
        res+=","+n[k];
    }
    res+="]";
    return res;
}
static void Main(){
    int A=100;
    Console.WriteLine("До вызова метода alpha(): A="+A);
    alpha(ref A);
    Console.WriteLine("После вызова метода alpha(): A="+A);
```

```
        int[] B={1,3,5};
        Console.WriteLine("До вызова метода bravo(): B="+ArrayToText(B));
        bravo(ref B);
        Console.WriteLine("После вызова метода bravo(): B="+ArrayToText(B));
        int[] C={2,4,6};
        Console.WriteLine("До вызова метода charlie(): C="+ArrayToText(C));
        charlie(ref C);
        Console.WriteLine("После вызова метода charlie(): C="+ArrayToText(C));
    }
}
```

В этом случае результат выполнения программы такой (жирным шрифтом выделены места, на которые стоит обратить внимание).

Результат выполнения программы (из листинга 5.6)

```
До вызова метода alpha(): A=100
В методе alpha(). На входе: 100
В методе alpha(). На выходе: 101
После вызова метода alpha: A=101
До вызова метода bravo(): B=[1,3,5]
В методе bravo(). На входе: [1,3,5]
В методе bravo(). На выходе: [2,4,6]
После вызова метода bravo(): B=[2,4,6]
До вызова метода charlie(): C=[2,4,6]
В методе charlie(). На входе: [2,4,6]
В методе charlie(). На выходе: [3,5,7]
После вызова метода charlie(): C=[3,5,7]
```

Видим, что теперь изменяется не только массив, на который ссылается переменная `B`, но и массив, на который ссылается переменная `C`, а также изменяется переменная `A`. Объяснение простое — аргументы передаются по ссылке, поэтому изменения, которые мы вносим в методе в аргументы, применяются к аргументам, а не к их копиям.

С передачей аргументов связан еще один механизм, позволяющий передавать в метод переменные, которым не присвоено значение — то есть

неинициализированные аргументы. Необходимость в использовании такого подхода может возникнуть, когда при выполнении метода нужно вернуть не одно значение, а сразу несколько. Можно, конечно, эти значения реализовать в виде массива, но этот прием не всегда приемлем. Можно передать аргумент по ссылке. Но в таком случае есть одна проблема: данному аргументу необходимо предварительно присвоить значение. А это тоже не всегда приемлемо. В таких случаях можно использовать передачу неинициализированного аргумента. То есть мы можем передать аргументом переменную, которая объявлена в программе, но значение ей не присваивалось. Просто так это сделать не получится — при компиляции возникнет ошибка. Придется использовать специальный режим. Если мы предполагаем, что некоторый аргумент метода будет передаваться через неинициализированную переменную, то соответствующий аргумент в описании метода указывается с идентификатором `out`. При вызове метода такой аргумент также передается с использованием идентификатора `out`. Есть еще одно важное условие: аргументу, который передан с `out`-инструкцией, при выполнении метода должно присваиваться значение.

Как иллюстрацию к использованию механизма передачи неинициализированных аргументов рассмотрим программу в листинге 5.7. Там описывается метод, который для массива, переданного аргументом, возвращает значение наименьшего элемента в массиве. Массив передается первым аргументом методу. А вторым аргументом методу передается переменная (неинициализированная), в которую записывается значение индекса элемента с наименьшим значением. Проанализируем код программы (жирным шрифтом выделены места, где используется инструкция `out`).

Листинг 5.7. Неинициализированный аргумент

```
using System;
class UsingOutDemo{
   // Метод для вычисления значения наименьшего элемента
   // в массиве и его индекса:
   static int getMin(int[] nums, out int index){
      // Начальное значение для индекса:
      index=0;
      // Перебор элементов массива:
      for(int k=1; k<nums.Length; k++){
```

```
            // Если значение элемента массива меньше текущего
            // минимального значения:
            if(nums[k]<nums[index]){
               // Новое значение для индекса:
               index=k;
            }
         }
         // Результат метода:
         return nums[index];
      }
      // Главный метод программы:
      static void Main(){
         // Целочисленный массив:
         int[] A={12,7,8,3,8,4,1,3,4,1,7,15};
         // Отображение содержимого массива:
         foreach(int v in A){
            Console.Write("| {0}", v);
         }
         Console.WriteLine("|");
         // Объявление переменных:
         int val, k;
         // Вычисление элемента с наименьшим значением:
         val=getMin(A, out k);
         // Отображение результатов:
         Console.WriteLine("Наименьшее значение: "+val);
         Console.WriteLine("Индекс элемента: "+k);
         Console.WriteLine("Проверка: A[{0}]={1}", k, A[k]);
      }
}
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 5.7)

```
| 12 | 7 | 8 | 3 | 8 | 4 | 1 | 3 | 4 | 1 | 7 | 15 |
```

```
Наименьшее значение: 1
Индекс элемента: 6
Проверка: A[6]=1
```

Метод для вычисления значения наименьшего элемента описан достаточно просто. Он возвращает `int`-значение, и у него два аргумента: целочисленный массив `nums` и целочисленный аргумент `index`, описанный с `out`-инструкцией. В теле метода командой `index=0` аргументу `index` присваивается начальное нулевое значение (индекс первого элемента в массиве). Затем перебираются прочие элементы массива (начиная со второго). Каждый очередной элемент сравнивается со значением элемента, чей индекс записан в аргумент `index` (условие `nums[k]<nums[index]` в условном операторе). Если значение проверяемого элемента с индексом `k` меньше значения элемента с индексом `index`, то командой `index=k` аргумент `index` получает новое значение.

После завершения выполнения оператора цикла в аргумент `index` записано значение элемента массива с наименьшим значением. Если таких элементов несколько, то запоминается индекс первого из них. Результат метода возвращается командой `return nums[index]`. В данном случае мы по индексу элемента определяем значение элемента, и это значение является результатом метода.

В главном методе программы командой `int[] A={12,7,8,3,8,4,1,3,4,1,7,15}` создается и инициализируется целочисленный массив. Содержимое массива отображается в консольном окне. Также в программе объявляются две целочисленные переменные `val` и `k`. Несмотря на то что переменной `k` значение не присваивалось, выполняется команда `val=getMin(A, out k)`. Второй аргумент методу `getMin()` передается с `out`-инструкцией. После выполнения команды в переменную `val` записывается значение элемента с наименьшим значением, а в переменную `k` записывается значение индекса этого элемента (первого из них). Значение переменных `val` и `k` отображается в консольном окне. Для проверки мы также отображаем значение элемента из массива `A` с индексом `k` (элемент `A[k]`). Понятно, что значения переменной `val` и элемента `A[k]` должны совпадать.

НА ЗАМЕТКУ

Описанные выше механизмы передачи аргументов применяются ко всем методам, не только статическим.

Рекурсия

Именно так выражается ее потребность в мировой гармонии.

из к/ф «Покровские ворота»

При описании методов можно использовать *рекурсию*. При рекурсии в теле метода вызывается этот же метод (но обычно с другим значением аргумента или аргументов). Такой подход позволяет создавать эффектные коды. Правда, «эффектный» не всегда означает «эффективный». Классическим примером использования рекурсии является программа для вычисления факториала числа (с использованием рекурсии). Чтобы понять, как организуется программный код, имеет смысл учесть, что факториал числа $n! = 1 \times 2 \times 3 \times ... \times (n-1) \times n$ может быть представлен в виде $n! = (n-1)! \times n$ — то есть факториал для числа n может быть представлен как факториал для числа $n-1$, умноженный на n. Программный код статического метода, вычисляющего факториал числа (переданного аргументом методу), может быть таким:

```
static int factorial(int n){
   if(n==1) return 1;
   else return n*factorial(n-1);
}
```

Основу тела метода `factorial()` составляет условный оператор, в котором проверяется равенство единице аргумента метода `n` (условие `n==1`). Если это так, то результатом метода возвращается значение `1` (по определению 1! = 1). Если условие ложно, то результатом метода возвращается значение выражения `n*factorial(n-1)`. В этом выражении использован вызов метода `factorial()`, но с аргументом `n-1`. Как это работает? Достаточно просто. Допустим, метод `factorial()` вызывается с аргументом `1`. Тогда при проверке условия в условном операторе оно окажется истинным и результатом метода возвращается значение `1`. Теперь предположим, что метод `factorial()` вызывается с аргументом, отличным от `1` — например, с аргументом `3`. При проверке условия оно окажется ложным, и результатом метода возвращается значение выражения `3*factorial(2)`. Но перед тем как вернуть значение, его необходимо вычислить. А для этого снова вызывается метод `factorial()`, но с аргументом `2`. Проверяется условие, оно ложное, и значение выражения `factorial(2)` вычисляется как `2*factorial(1)`. Снова вызывается метод `factorial()`, но с аргументом `1`. А мы уже знаем, что

значением выражения `factorial(1)` является `1`. Тогда значение выражения `factorial(2)` равно `2`, а значение выражения `factorial(3)` равно `6`. Понятно, что, чем больше значение аргумента метода, тем длиннее цепочка вызовов метода `factorial()`.

Аналогичным образом с использованием рекурсии описываются и другие методы. Например, так может выглядеть программный код для статического метода, вычисляющего число в последовательности Фибоначчи по его номеру (аргумент метода):

```
static int fibs(int n){
   if(n==1||n==2) return 1;
   else return fibs(n-1)+fibs(n-2);
}
```

Здесь мы учли тот факт, что при значениях аргумента `1` или `2` (первое или второе число в последовательности) метод должен возвращать значение `1`. Во всех остальных случаях число с номером `n` вычисляется как сумма двух предыдущих чисел. Поэтому программный код метода `fibs()` организован в виде условного оператора, в котором проверяется условие `n==1||n==2` (аргумент `n` равен `1` или `2`). Если так, то результатом метода возвращается значение `1`. Если условие ложно, то результатом метода возвращается значение выражения `fibs(n-1)+fibs(n-2)`, в котором вызывается метод `fibs()` (причем дважды).

Рекурсию можно использовать даже там, где ее, на первый взгляд, не может быть вообще. Например, задача на вычисление суммы натуральных чисел. Допустим, нам нужно вычислить сумму $S(n) = 1 + 2 + 3 + \ldots + (n-1) + n$. Но ее можно представить в виде $S(n) = n + S(n-1)$ — то есть сумма n натуральных чисел есть сумма $n-1$ натуральных чисел плюс число n. Значит, статический метод для вычисления суммы натуральных чисел с использованием рекурсии можно описать так:

```
static int sum(int n){
   if(n==0) return 0;
   else return n+sum(n-1);
}
```

То есть если аргумент метода равен `0`, то результатом является значение `0`. В противном случае значение выражения `sum(n)` вычисляется как `n+sum(n-1)`. В листинге 5.8 представлена программа со статическими методами, в описании которых использована рекурсия.

Листинг 5.8. Использование рекурсии

```
using System;
class RecursionDemo{
   // Метод для вычисления факториала числа:
   static int factorial(int n){
      if(n==1) return 1;
      else return n*factorial(n-1);
   }
   // Метод для вычисления чисел Фибоначчи:
   static int fibs(int n){
      if(n==1||n==2) return 1;
      else return fibs(n-1)+fibs(n-2);
   }
   // Метод для вычисления суммы чисел:
   static int sum(int n){
      if(n==0) return 0;
      else return n+sum(n-1);
   }
   // Метод для отображения содержимого массива:
   static void show(int[] a, int k){
      // Отображение значения элемента массива:
      Console.Write(a[k]+" ");
      // Если элемент в массиве последний:
      if(k==a.Length-1){
         Console.WriteLine();
      }
      // Если элемент в массиве не последний:
      else{
         // Рекурсивный вызов метода:
         show(a, k+1);
      }
   }
```

```
   // Перегрузка метода для отображения
   // содержимого массива:
   static void show(int[] a){
      // Вызов версии метода с двумя аргументами:
      show(a,0);
   }
   // Главный метод программы:
   static void Main(){
      Console.WriteLine("Факториал числа:");
      for(int k=1; k<=10; k++){
         // Вычисление факториала числа:
         Console.WriteLine(k+"!="+factorial(k));
      }
      Console.WriteLine("Числа Фибоначчи:");
      for(int k=1; k<=10; k++){
         // Вычисление чисел Фибоначчи:
         Console.Write(fibs(k)+" ");
      }
      Console.WriteLine();
      Console.Write("Сумма чисел от 1 до 100: ");
      Console.WriteLine(sum(100));
      // Числовой массив:
      int[] A={1,3,5,7,9,11,13,15,17,19,21};
      Console.WriteLine("Числовой массив:");
      // Отображение всех элементов массива:
      show(A);
      Console.WriteLine("Элементы, начиная с третьего:");
      // Отображение элементов, начиная с третьего:
      show(A,2);
   }
}
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 5.8)

```
Факториал числа:
1!=1
2!=2
3!=6
4!=24
5!=120
6!=720
7!=5040
8!=40320
9!=362880
10!=3628800
Числа Фибоначчи:
1 1 2 3 5 8 13 21 34 55
Сумма чисел от 1 до 100: 5050
Числовой массив:
1 3 5 7 9 11 13 15 17 19 21
Элементы, начиная с третьего:
5 7 9 11 13 15 17 19 21
```

В программе, кроме уже описанных методов, использовался метод `show()` для отображения массива. В нем также задействована рекурсия. Аргументов у метода два: отображаемый массив (объявлен как `int[] a`) и индекс элемента, начиная с которого выполняется отображение элементов массива (соответствующий аргумент объявлен как `int k`). В теле метода командой `Console.Write(a[k]+" ")` отображается значение элемента массива с индексом, переданным вторым аргументом методу. Затем в условном операторе проверяется условие `k==a.Length-1`. Если оно истинно, то это означает, что мы имеем дело с последним элементом в массиве. В таком случае командой `Console.WriteLine()` выполняется переход к новой строке. Но если условие оказалось ложным, то командой `show(a, k+1)` выполняется рекурсивный вызов метода `show()`. Первый аргумент, переданный методу, — все тот же массив. А второй аргумент — на единицу больше. Получается, что когда вызывается метод `show()`, то аргументом ему передается массив и индекс элемента. Если это индекс последнего элемента, то его

значение отображается в консольном окне и, в общем-то, все (за исключением перехода к новой строке). Если элемент не последний, то после отображения его значения метод `show()` вызывается для следующего элемента, и так далее по цепочке, пока метод не будет вызван с индексом последнего элемента.

Если мы хотим отобразить значения всех элементов массива, то нам необходимо вызвать метод `show()` со вторым нулевым аргументом. Мы для удобства переопределяем метод `show()`, описав версию метода без второго аргумента (первый аргумент — отображаемый массив). В описании версии метода `show()` с одним аргументом есть всего одна команда `show(a,0)`, которой вызывается версия метода `show()` с двумя аргументами, причем второй аргумент нулевой.

ПОДРОБНОСТИ

При перегрузке метода `show()` о рекурсии речь не идет. Дело в том, что когда в теле версии метода `show()` с одним аргументом вызывается версия метода `show()` с двумя аргументами, то фактически вызывается другой метод (просто название у него такое же). Это не рекурсивный вызов. При рекурсивном вызове метод вызывает сам себя.

В главном методе программы проиллюстрирована работа статических методов. В частности, там для чисел от 1 до 10 вычисляется факториал, вычисляется последовательность чисел Фибоначчи, а также отображается содержимое одномерного целочисленного массива. Во всех этих случаях используются методы, реализованные на основе рекурсии.

Методы с произвольным количеством аргументов

> История, леденящая кровь. Под маской овцы скрывался лев!
>
> *из к/ф «Покровские ворота»*

Допустим, мы хотим описать метод, который вычисляет сумму переданных ему аргументов. В принципе, путем перегрузки метода мы можем описать версию с одним аргументом, двумя аргументами, тремя аргументами и так далее. Проблема в том, что на каждое количество аргументов нужна «персональная» версия метода. И сколько бы версий метода мы ни описали, всегда

можно передать на один аргумент больше. Здесь мы сталкиваемся с проблемой, которая состоит в том, что наперед не известно, сколько аргументов будет передано методу при вызове. Причем мы хотим, чтобы можно было передавать любое количество. Именно для подобных случаев в C# есть механизм, позволяющий описывать методы с произвольным количеством аргументов. Набор аргументов, количество которых неизвестно, формально описывается как массив, но только перед описанием этого массива в списке аргументов указывается ключевое слово `params`. В теле метода обрабатываются такие аргументы, как элементы массива. Но при вызове метода аргументы просто перечисляются через запятую. Например, если некоторый метод описан в формате `метод(params int[] args)`, то это означает, что при вызове методу может быть передано произвольное количество целочисленных аргументов. В теле метода к этим аргументам обращаемся так, как если бы они были элементами массива `args`: то есть `args[0]` (первый аргумент), `args[1]` (второй аргумент) и так далее. Количество переданных аргументов можно определить с помощью свойства `Length` (выражение вида `args.Length`).

НА ЗАМЕТКУ

В принципе, вместо аргумента, описанного как массив с ключевым словом `params`, можно передавать обычный массив соответствующего типа.

Метод можно описывать так, что в нем будут как «обычные» (явно описанные) аргументы, так и аргументы, количество которых наперед не задано. В таком случае «обычные» аргументы описываются сначала. Допустим, мы хотим описать метод, которому при вызове обязательно передаются один символьный аргумент и произвольное количество целочисленных аргументов. Такой метод описывается в формате `(char symb, params int[] args)`. Нельзя описать метод, у которого несколько наборов аргументов, количество которых неизвестно. Например, не получится описать метод, которому передается произвольное количество символьных и целочисленных аргументов.

ПОДРОБНОСТИ

При вызове метода фактически переданные ему аргументы заносятся в память. Если известен объем памяти, занимаемый аргументами, и тип значений, то можно вычислить количество этих аргументов. Если, например, известно, что методу передается один символьный

аргумент и произвольное количество целочисленных аргументов, то задача о «распределении памяти» решается однозначно — можно определить, какая память выделена под символьный аргумент, а то, что осталось, выделено под целочисленные аргументы. Зная, какой объем памяти выделяется для значения целочисленного типа, можно определить количество целочисленных аргументов и область памяти, выделенную для каждого из них. Но если бы было указано, что некоторая область памяти выделена для неизвестного количества символьных и целочисленных значений, то здесь могут быть разные варианты распределения памяти. Проще говоря, задача не имеет однозначного решения. Поэтому в методе может быть только один набор аргументов, количество которых не задано.

Небольшая программа, иллюстрирующая способы описания и использования методов с произвольным количеством аргументов, представлена в листинге 5.9.

Листинг 5.9. Методы с произвольным количеством аргументов

```
using System;
class ParamsDemo{
   // Метод для вычисления суммы чисел:
   static int sum(params int[] a){
      // Локальная переменная (значение суммы):
      int res=0;
      // Перебор аргументов метода:
      for(int k=0; k<a.Length; k++){
         // Прибавление слагаемого к сумме:
         res+=a[k];
      }
      // Результат метода:
      return res;
   }
   // Метод для извлечения символов из текста:
   static string getText(string t, params int[] a){
      // Начальное значение формируемой текстовой строки:
      string res="";
```

```
    // Перебор аргументов метода:
    for(int k=0; k<a.Length; k++){
       // Добавление символа в текст:
       res+=t[a[k]];
    }
    // Результат метода:
    return res;
}
// Метод отображает значения аргументов:
static void show(int[] a, params char[] b){
    // Количество элементов в числовом массиве:
    Console.Write("Чисел "+a.Length+": ");
    // Значения элементов в числовом массиве:
    for(int k=0; k<a.Length-1; k++){
       Console.Write(a[k]+" ");
    }
    Console.WriteLine("и "+a[a.Length-1]);
    // Количество символьных аргументов:
    Console.Write("Символов "+b.Length+": ");
    // Значения символьных аргументов:
    for(int k=0; k<b.Length-1; k++){
       Console.Write(b[k]+" ");
    }
    Console.WriteLine("и "+b[b.Length-1]);
}
// Главный метод программы:
static void Main(){
    // Примеры вызова методов.
    // Вычисление суммы чисел:
    Console.WriteLine("Сумма чисел: "+sum(1,6,9,2,4));
    Console.WriteLine("Сумма чисел: "+sum(5,1,2));
    // Формируется текст:
```

```
        Console.WriteLine(getText("Раз два три",0,10,8,1));
        Console.WriteLine(getText("Бревно",3,5,1,5,4));
        // Отображаются аргументы:
        show(new int[]{1,3,5},'A','B','C','D','E');
        show(new int[]{1,3,5,7,9},'A','B','C','D');
    }
}
```

Как будет выглядеть результат выполнения этой программы, показано ниже.

Результат выполнения программы (из листинга 5.9)

```
Сумма чисел: 22
Сумма чисел: 8
Рита
ворон
Чисел 3: 13 и 5
Символов 5: A B C D и E
Чисел 5: 1357 и 9
Символов 4: A B C и D
```

В программе описан статический метод `sum()`, предназначенный для вычисления суммы чисел, переданных аргументами методу. Аргумент описан как `params int[] a`. В теле метода объявлена локальная переменная `res` с начальным нулевым значением. В операторе цикла перебираются элементы массива `a` (но в реальности это перебор аргументов, переданных методу), и за каждый цикл командой `res+=a[k]` к текущему значению переменной `res` прибавляется значение очередного аргумента. После того как сумма аргументов вычислена и записана в переменную `res`, значение переменной возвращается результатом метода. Так, если вызвать метод командой `sum(1,6,9,2,4)`, то значением будет число `22` (сумма чисел `1`, `6`, `9`, `2` и `4`). Значением выражения `sum(5,1,2)` является число `8` (сумма чисел `5`, `1` и `2`).

Метод `getText()` предназначен для формирования одной текстовой строки на основе другой текстовой строки. Аргументом методу передается текст (первый аргумент) и произвольное количество целочисленных

аргументов. Эти целочисленные аргументы определяют индексы символов в текстовой строке (первый аргумент), из которых следует сформировать строку-результат. Например, если метод вызывается командой `getText("Раз два три",0,10,8,1)`, то значение вычисляется так: из строки `"Раз два три"` берем символы с индексами `0`, `10`, `8` и `1` и объединяем их в текстовую строку. Это слово `"Рита"`. Если метод вызывается командой `getText("Бревно",3,5,1,5,4)`, то результатом является текст `"ворон"`: из текста `"Бревно"` следует взять символы с индексами `3`, `5`, `1`, снова `5` и `4`.

Код метода `getText()` реализован несложно. Объявляется текстовая переменная `res`, начальное значение которой — пустая текстовая строка. Затем запускается оператор цикла, в котором перебираются целочисленные аргументы метода (отождествляются с элементами массива `a`). Командой `res+=t[a[k]]` к текущему значению текстовой переменной `res` дописывается символ из текста `t` (первый аргумент метода) с индексом `a[k]`. Строка `res` возвращается результатом метода.

Метод `show()` выполняет незатейливую работу — он отображает значения переданных ему аргументов. Первым аргументом методу передается целочисленный массив `a` (обычный). После этого обязательного аргумента методу может передаваться произвольное количество символьных аргументов. Поэтому формальный второй аргумент `b` метода описан как массив с идентификатором `params`. Методом при вызове отображается такая информация:

- количество элементов в числовом массиве (определяется выражением `a.Length`);
- значения элементов в числовом массиве, причем перед последним значением отображается текст `"и "`;
- количество символьных аргументов (определяется выражением `b.Length`);
- значения символьных аргументов (перед последним значением отображается текст `"и "`).

Для проверки работы метода использованы команды `show(new int[]{1,3,5},'A','B','C','D','E')` и `show(new int[]{1,3,5,7,9},'A','B','C','D')`. В обоих случаях первым аргументом методу передаются анонимные массивы, которые создаются командой вида `new int[]{значения}`.

НА ЗАМЕТКУ

Например, командой `new int[]{1,3,5}` создается массив из трех элементов со значениями `1`, `3` и `5`. Значение выражения `new int[]{1,3,5}` — это ссылка на созданный массив. Эта ссылка передается первым аргументом методу `show()`.

Главный метод программы

> Известный чародей и магистр тайных сил. Нынче в Петербурге шуму много наделал. Газеты пишут — камни драгоценные растил, будущность предсказывал.
>
> *из к/ф «Формула любви»*

Главный метод программы может возвращать результат. Возвращаемый главным методом результат является индикатором того, что работа программы завершилась в нормальном режиме (без ошибок). Поскольку результат предназначен для исполнительной системы, то в прикладном плане все сводится к возможности описывать главный метод программы способом, отличным от того, который мы использовали ранее и будем использовать впредь.

Итак, вместо того чтобы описывать метод `Main()` как не возвращающий результат (с идентификатором `void`), метод `Main()` можно описать как возвращающий целочисленное значение (тип результат `int`). При этом в теле метода должна быть `return`-инструкция, которая, собственно, результат и возвращает. Возвращающим значением обычно является `0`, означающий, что выполнение программы завершилось штатно. Как правило, если метод `Main()` описан с идентификатором `int` для типа результата, то последней командой в теле метода является инструкция `return 0`. Как выглядит программа в таком случае, показано в листинге 5.10 (жирным шрифтом выделены места кода, на которые стоит обратить внимание).

Листинг 5.10. Главный метод возвращает результат

```
using System;
class MainMethDemo{
```

```
   // Главный метод возвращает результат:
   static int Main(){
      Console.WriteLine("Главный метод возвращает результат!");
      // Результат главного метода:
      return 0;
   }
}
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 5.10)

```
Главный метод возвращает результат!
```

Еще раз подчеркнем, что особенность данной программы в том, что метод `Main()` описан с идентификатором `int` и содержит `return`-инструкцию. При желании можно описывать главный метод в представленном выше формате.

НА ЗАМЕТКУ

Мы уже знаем, что главный метод программы может описываться с аргументом — текстовым массивом, через который в программу передаются параметры командной строки. Также нередко главный метод программы описывают с ключевым словом `public`. Все эти варианты описания метода `Main()`, включая реализацию метода, возвращающего целочисленный результат, можно «комбинировать». Например, мы можем описать главный метод программы с ключевым словом `public` как такой, что возвращает `int`-значение, и с аргументом — текстовым массивом.

Резюме

Мастера не мудрствуют.

из к/ф «Покровские ворота»

- Метод — именованный блок кода, который может выполняться многократно (через вызов метода). Методы бывают статические и нестатические. Для вызова нестатического метода нужен объект. Для вызова статического метода объект не нужен.

- Статический метод описывается с ключевым словом `static`. После него указывается идентификатор типа результата, возвращаемого методом (если метод не возвращает результат — указывают идентификатор `void`). Далее следует имя метода, в круглых скобках описываются аргументы метода, а код метода описывается в блоке из фигурных скобок. Значение, возвращаемое методом, в теле метода указывают после инструкции `return`.
- Методы можно перегружать: в таком случае описывается несколько версий метода с одним и тем же именем. Перегруженные версии метода должны отличаться количеством и/или типом аргументов. Решение о том, какую версию следует вызывать, принимается на основе аргументов, которые фактически переданы методу.
- При передаче аргументом методу массива в действительности в метод передается ссылка на массив. Если метод возвращает результатом массив, то обычно при вызове метода создается массив, а ссылка на него возвращается результатом метода.
- Аргументы в метод могут передаваться по значению и по ссылке. При передаче аргументов по значению (такой режим используется по умолчанию) для аргументов создаются технические копии и все операции выполняются с ними. При передаче аргументов по ссылке в метод передаются те переменные, которые указаны аргументами. Чтобы аргумент передавался по ссылке, в описании метода аргумент должен быть описан с инструкцией `ref`. Такая же инструкция указывается вместе с аргументом при вызове метода.
- В метод можно передавать неинициализированный аргумент — переменную, которая объявлена, но не инициализирована. Такой аргумент должен получить значение в процессе выполнения метода. Для возможности передачи неинициализированного аргумента в описании метода такой аргумент описывается с инструкцией `out`. Инструкция `out` также указывается вместе с неинициализированным аргументом при вызове метода.
- При описании методов можно использовать рекурсию. При рекурсии в процессе выполнения метода он вызывает сам себя.
- Можно описывать методы с аргументами, количество которых заранее не известно. Такие параметры формально описываются как массив с инструкцией `params`. В теле метода аргументы обрабатываются как элементы массива, а при вызове метода аргументы передаются

как обычно — через запятую. Если у метода, кроме набора аргументов неизвестного количества, есть и «обычные» аргументы, то они в описании метода указываются в начале. Аргумент с инструкцией `params` должен быть последним в списке аргументов метода.

- Главный метод программы `Main()` может описываться как такой, что возвращает `int`-значение. В таком случае в теле метода обычно последней является инструкция `return 0`.

Задания для самостоятельной работы

Я нришел вам сделать предложение впрок.

из к/ф «Старики-разбойники»

1. Напишите программу, в которой описан статический метод для вычисления двойного факториала числа, переданного аргументом методу. По определению, двойной факториал числа n (обозначается как $n!!$) — это произведение через одно всех чисел, не больших числа n. То есть $n!! = n \times (n-2) \times (n-4) \times \ldots$ (последний множитель равен 1 для нечетного n и равен 2 для четного n). Например, $6!! = 6 \times 4 \times 2 = 48$ и $5!! = 5 \times 3 \times 1 = 15$. Предложите версию метода без рекурсии и с рекурсией.

2. Напишите программу со статическим методом, которым вычисляется сумма квадратов натуральных чисел $1^2 + 2^2 + 3^2 + \ldots + n^2$. Число n передается аргументом методу. Предложите версию метода с рекурсией и без рекурсии. Для проверки результата можно использовать формулу $1^2 + 2^2 + 3^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$.

3. Напишите программу со статическим методом, которому аргументом передается целочисленный массив и целое число. Результатом метод возвращает ссылку на новый массив, который получается из исходного массива (переданного первым аргументом методу), если в нем взять несколько начальных элементов. Количество элементов, которые нужно взять из исходного массива, определяются вторым аргументом метода. Если второй аргумент метода больше длины массива, переданного первым аргументом, то методом создается копия исходного массива и возвращается ссылка на эту копию.

4. Напишите программу со статическим методом, аргументом которому передается символьный массив, а результатом возвращается ссылка

на целочисленный массив, состоящий из кодов символов из массива-аргумента.

5. Напишите программу со статическим методом, аргументом которому передается целочисленный массив, а результатом возвращается среднее значение для элементов массива (сумма значений элементов, деленная на количество элементов в массиве).

6. Напишите программу со статическим методом, аргументом которому передается двумерный целочисленный массив. У метода, кроме аргумента-массива, есть два неинициализированных аргумента. Результатом метод возвращает значение наибольшего элемента в массиве. Неинициализированным аргументам присваиваются значения индексов этого элемента.

7. Напишите программу со статическим методом, аргументом которому передается одномерный символьный массив. В результате вызова метода элементы массива попарно меняются местами: первый — с последним, второй — с предпоследним и так далее.

8. Напишите программу с перегруженным статическим методом. Если аргументом методу передается два целых числа, то результатом возвращается ссылка на целочисленный массив, состоящий из натуральных чисел, а первое и последнее число в массиве определяется аргументами метода. Например, если передать аргументами числа `2` и `4`, то результатом будет массив из чисел `2`, `3` и `4`. Если аргументами методу передаются два символьных значения, то результатом возвращается ссылка на массив, состоящий из последовательности символов, а первый и последний символы определяются аргументами метода. Например, если передать аргументами методу символы `'B'` и `'D'`, то в результате получим массив из символов `'B'`, `'C'` и `'D'`.

9. Напишите программу со статическим методом, аргументом которому передается произвольное количество целочисленных аргументов. Результатом метод возвращает массив из двух элементов: это значения наибольшего и наименьшего значений среди аргументов, переданных методу.

10. Напишите программу со статическим методом, которому передается текст и произвольное количество символьных аргументов. Результатом возвращается текст, который получается добавлением в конец исходного текста (первый аргумент метода) символьных значений, переданных аргументами методу.

Глава 6
ЗНАКОМСТВО С КЛАССАМИ И ОБЪЕКТАМИ

Вообще-то я не специалист но этим гравицапам.

из к/ф «Кин-дза-дза»

В этой главе состоится наше первое осознанное знакомство с *классами* и *объектами*. Мы будем обсуждать следующие вопросы:

- познакомимся со способами описания классов;
- узнаем, что такое объекты и как они создаются;
- научимся описывать и использовать поля и методы;
- узнаем, какие операции и как выполняются с объектами;
- расширим наши представления о перегрузке методов;
- познакомимся с конструкторами и деструкторами.

Но прежде всего познакомимся с базовыми принципами *объектно ориентированного программирования*.

Базовые принципы ООП

Если у меня немножко КЦ есть, я имею право носить желтые штаны и нередо мной нацак должен не один, а два раза нриседать.

из к/ф «Кин-дза-дза»

Как мы уже знаем, язык программирования C# — полностью объектно ориентированный. Это означает, что в нем реализована парадигма ООП. В чем же особенность ООП? Важно понимать, что речь идет о способе организации программы. На сегодня существует два основных способа организации программного кода. Это объектно ориентированное

и процедурное (структурное) программирование. Попробуем разобраться, чем отличаются эти концепции программирования.

При процедурном программировании программа реализуется как набор подпрограмм (процедуры и функции), используемых для обработки данных. То есть данные, используемые в программе, и программный код, применяемый для обработки этих данных, существуют отдельно друг от друга. В процессе выполнения программы выбираются нужные данные и обрабатываются с помощью специальных процедур и функций. Если провести аналогию, то можно такой способ программирования и организации программы сравнить с кухней. На этой кухне стоят холодильники и специальные боксы с продуктами. Это данные. Еще на кухне есть повара, которые из продуктов готовят блюда. Повара — это аналог процедур и функций. Есть шеф-повар (которого можно отождествить с программой или, как в нашем случае, с главным методом). Шеф-повар определяет блюда, которые должны готовить его подчиненные, и какие продукты (из какого холодильника или бокса) должны использоваться. Что не так в этой схеме? Да, в общем-то, все нормально. Если шеф-повар профессиональный, а команда поваров подобрана грамотно, то обязанности будут распределены правильно и кухня будет работать нормально. Так все происходит, если кухня не очень большая. Но если у вас огромный цех, в котором много поваров и холодильников с продуктами, то могут возникнуть проблемы организационного характера: кто-то не в своем холодильнике продукты взял, кто-то заказ неправильно понял. При этом повара-исполнители могут быть высочайшими профессионалами и продукты могут быть отменного качества. Но этого мало. Важно еще и правильно организовать процесс. А если кухня большая, то сделать это непросто. Какой выход из ситуации? Реорганизация. Разбиваем весь большой кухонный цех на блоки, в каждом блоке есть свой сушеф (главный повар в блоке), у него есть подчиненные. У каждого блока есть своя специализация. Один блок готовит первые блюда, другой готовит салаты, третий готовит десерты, и так далее. В каждом блоке свои холодильники и боксы с продуктами, предназначенными именно для этого блока. Шеф-повар общается с сушефами в блоках. Им он раздает задания, а они уже контролируют работу своих блоков. То есть сушефы организуют и контролируют работу блока, а шеф-повар руководит работой кухни на уровне взаимодействия разных блоков. А что там происходит внутри каждого отдельного блока — не его забота. Собственно, это и есть объектно ориентированный подход. При таком подходе программа представляет собой «сцену», на которой взаимодействуют

«объекты». «Объектами» в нашем «кухонном» примере являются блоки. Каждый «объект» содержит в себе данные (холодильники с продуктами) и методы, обрабатывающие данные (повара, которые используют продукты), и все это локализовано в объекте. Главное удобство описанного подхода в том, что мы сразу распределяем и группируем данные и программный код, предназначенный для обработки этих данных. Это потенциально уменьшает вероятность ошибочного использования данных и облегчает создание программы.

Можно привести и другую аналогию для иллюстрации разницы между процедурным и объектно ориентированным программированием. Допустим, нам нужно построить дом. Мы его можем строить из кирпичей. Если дом не очень большой, то такая технология вполне удачная. Из кирпичей можно выложить практически все что угодно, реализовать любой дизайнерский проект. Но все это хорошо, пока мы не решили строить многоэтажный дом. Если так, то технологию, скорее всего, придется поменять. Намного удобнее строить такой дом из блоков. Каждый блок заранее изготавливается на специальном заводе. Блок содержит окна, двери и другие полезные элементы. И весь дом складывается из блоков. Это будет быстрее и надежнее по сравнению со случаем, когда мы строим из кирпичей.

Строительство дома из кирпичей — аналог программы, которая создается в рамках концепции процедурного программирования. Строительство дома из блоков — аналог программы, созданной в рамках концепции ООП. Мы в этом случае «строим» программу не из маленьких «кирпичиков», а из больших «блоков», которые имеют еще и некоторую функциональность.

Таким образом, ООП — это способ организации программы через взаимодействие отдельных объектов, содержащих данные и методы для работы с этими данными. Обычно в ООП выделяют три базовых принципа:

- инкапсуляция;
- полиморфизм;
- наследование.

Любой объектно ориентированный язык содержит средства для реализации этих принципов. Способы реализации могут отличаться, но принципы неизменны.

Инкапсуляция означает, что данные объединяются в одно целое с программным кодом, предназначенным для их обработки. Фактически организация программы через взаимодействие объектов является реализацией принципа инкапсуляции. На программном уровне инкапсуляция реализуется путем использования *классов* и *объектов* (объекты создаются на основе классов).

НА ЗАМЕТКУ

Классы и объекты обсуждаются в следующем разделе.

Полиморфизм подразумевает использование единого интерфейса для решения однотипных задач. Проявлением полиморфизма является тот факт, что нередко в программе один и тот же метод можно вызывать с разными аргументами. Это удобно. Если вернуться к аналогии с кухней, то ситуация примерно такая: шеф-повар дает задание своим поварам, указывая название блюда. При этом ему нет необходимости уточнять, как именно блюдо готовится. Все это знают повара, выполняющие заказ. Они определяют, какие продукты нужно использовать. Если в какой-то момент поменяется рецепт приготовления блюда (например, один продукт будет заменен другим), шеф-повар будет называть все то же блюдо. Изменение рецепта автоматически учтут повара, которые готовят блюдо.

НА ЗАМЕТКУ

Проявлением полиморфизма является перегрузка и переопределение методов, о чем речь еще будет идти.

Наследование позволяет создавать объекты не на пустом месте, а с использованием ранее разработанных утилит. В языке C# наследование позволяет создавать классы на основе уже существующих классов. Эти классы используются для создания объектов. Вообще название этого механизма очень точно отображает его сущность. Он занимает важное место в концепции языка C#, и мы обязательно вернемся к обсуждению этого вопроса.

НА ЗАМЕТКУ

На уровне «кухни» наследование могло бы быть реализовано следующим образом. Допустим, имеется отдел по изготовлению десертов. Мы хотим расширить возможности этого отдела так, чтобы там

готовили еще и коктейли (имеет ли это смысл на реальной кухне — в данном случае не важно). Какие есть варианты? Можно расформировать старый отдел и создать новый «с нуля», а можно оставить старый отдел, но добавить в него еще одно внутреннее «подразделение», которое будет заниматься коктейлями. В некотором смысле это напоминает наследование: берем то, что уже есть, и добавляем к нему нечто новое.

Далее мы перейдем к более приземленным вопросам. В частности, познакомимся с классами и объектами. Концепция классов и объектов является фундаментальной для понимания принципов реализации ООП в языке C#.

Классы и объекты

Дальше следует непереводимая игра слов с использованием местных идиоматических выражений.

из к/ф «Бриллиантовая рука»

В первую очередь нам следует понять, чем *класс* принципиально отличается от *объекта*. Для этого мы опять прибегнем к аналогии. Представим, что имеется автомобильный завод, с конвейера которого выходят автомобили. А где-то есть главный инженер, и у него имеется план или чертеж, по которому эти автомобили собираются.

НА ЗАМЕТКУ

В действительности все не так, все намного сложнее — но здесь это неважно. Нам нужно понять базовые принципы, а для этого вполне подойдет сильно упрощенная схема производства автомобилей.

Так вот, чертеж или план, на котором изображен автомобиль со всеми его техническими характеристиками, — это аналог класса. А реальные автомобили, которые выходят с конвейера и которые собраны в соответствии с чертежом, — аналог объектов. Все автомобили (если они собраны по одному и тому же чертежу) — однотипные. У них одинаковый набор характеристик, и в плане функциональности они эквивалентны. Но физически автомобили различны, у каждого своя «судьба».

НА ЗАМЕТКУ

У автомобилей, собранных по одному чертежу, одинаковый набор характеристик, но значения некоторых характеристик могут отличаться. Например, у каждого автомобиля есть такая характеристика, как цвет. Но значение этой характеристики у разных автомобилей может быть разным: скажем, один автомобиль красный, а другой — синий.

Если обратиться к примеру с постройкой дома, то там аналогом объектов являются блоки, из которых строится дом. Базовый чертеж, на основе которого на заводе изготовлялся блок, является аналогом класса. Допустим, дом строится с использованием блоков пяти разных типов. Это означает, что имеется пять разных чертежей блоков, на основании которых создаются блоки. Если блоки создавались на основе разных чертежей, то они будут разными. Если блоки создавались на основе одного и того же чертежа, то они будут одинаковыми (точнее, однотипными).

НА ЗАМЕТКУ

В известном смысле, класс можно рассматривать как некий тип данных. Но только это специфический тип, поскольку кроме собственно данных в нем «спрятана» еще и некоторая функциональность (способность выполнять определенные действия).

Наличие чертежа не означает наличия блока. Блоки нужно создавать. На основании одного чертежа можно создать много блоков. Но также может быть и ситуация, когда чертеж есть, а блоки по нему не создавались. Это же справедливо для классов и объектов. Класс нужен для создания на его основе объектов. На основе одного класса можно создать много объектов. А можно описать класс и не создавать объектов совсем. Все зависит от решаемой задачи.

Аналогии — это хорошо. Но что же такое объект, если речь идет об использовании его в программе? Скажем, мы уже знаем, что за переменной «скрывается» область в памяти. В эту область можно записывать значения и считывать значение оттуда. И для получения доступа к этой памяти достаточно знать имя переменной. Объект во многом похож на переменную, но только он более «разноплановый». Объект — это группа переменных, причем в общем случае разного типа. А еще объект — это набор методов. Метод, в свою очередь — это группа инструкций, которые можно выполнить (вызвав метод).

НА ЗАМЕТКУ

Здесь и далее, если явно не указано противное, имеются в виду обычные, не статические методы. Обычный метод, в отличие от статического, «привязан» к объекту и не может быть вызван, если не указан объект, из которого вызывается метод.

В итоге получается, что если обычная переменная напоминает коробку со значением внутри, то объект — это большая коробка, в которой есть коробки поменьше (переменные) и всякие пружинки и рычажки (наборы инструкций, реализованные в виде методов). С программной точки зрения объект реализуется как некоторая область памяти, содержащая переменные и методы.

ПОДРОБНОСТИ

Откровенно говоря, в памяти коды методов для объектов обычно хранятся отдельно от переменных, входящих в объект. Связано это с оптимизацией. Но для понимания принципов работы объектов и использования их в программе можно полагать, что и переменные, и методы объекта находятся в «одной коробке».

Программная «начинка» объекта состоит из переменных и методов. Переменные называются полями объекта. Объекты, как мы помним, создаются на основе класса. Когда мы описываем класс, мы фактически определяем, какие поля и методы будут у объектов, создаваемых на основе класса.

Поля метода можно использовать как обычные переменные. Методы объекта можно вызывать. В этом случае выполняются инструкции, содержащиеся в теле метода. Метод, который вызывается из некоторого объекта, автоматически получает доступ к полям (и другим методам) этого объекта. Также важно помнить, что и поля, и методы у каждого объекта свои. Точно так же, как у каждого автомобиля свой руль, свои колеса и своя коробка передач. И если вы поворачивает ключ зажигания, то заведется тот автомобиль, в котором вы поворачиваете ключ. А если вы перекрасите свой автомобиль, то это никак не скажется на цвете автомобиля вашего соседа (даже если у него автомобиль той же марки).

Описание класса и создание объекта

> Как он может своими мозгами играть, когда он эти куклы нервый раз видит?
>
> *из к/ф «Кин-дза-дза»*

Нам предстоит выяснить как минимум два момента. Во-первых, нам нужно узнать, как описывается класс. И, во-вторых, мы должны научиться создавать на основе классов объекты (и использовать их в программе). Начнем с описания класса.

Описание класса начинается с ключевого слова `class`. После ключевого слова `class` указывается название класса. Описание класса размещается в блоке, выделенном фигурными скобками. Ниже приведен шаблон, в соответствии с которым описывается класс (жирным шрифтом выделены ключевые элементы шаблона):

```
class имя{
   // Описание класса
}
```

В теле класса описываются поля (переменные) и методы. Поля и методы класса называются *членами* класса. Поля описываются точно так же, как мы ранее объявляли переменные в главном методе программы: указывается тип поля и его название. Если несколько полей относятся к одному типу, то можно их объявить одной инструкцией, указав тип полей и перечислив их названия через запятую.

Методы описываются так:

- указывается тип результата;
- название метода;
- перечисляются аргументы (в круглых скобках — указывается тип и название каждого аргумента);
- команды, выполняемые при вызове метода, размещаются в блоке, выделенном фигурными скобками (тело метода).

ПОДРОБНОСТИ

Обычный (не статический) метод (а мы рассматриваем именно такие методы) вызывается из объекта. Этот метод имеет доступ к полям

объекта. То есть в описании метода можно обращаться к полям объекта, и при этом данные поля как-то дополнительно идентифицировать не нужно (достаточно их описания в теле класса). Метод может возвращать значение (если метод не возвращает значение, то идентификатором типа результата указывается ключевое слово `void`). В таком случае это значение отождествляется с инструкцией вызова метода. Эту инструкцию можно использовать как операнд в выражениях, подразумевая под ней результат, возвращаемый методом. Также методу могут передаваться аргументы. Это значения, которые метод использует при вызове. Аргументы описываются в круглых скобках после имени метода. Для каждого аргумента указывается тип. При вызове метода фактически передаваемые методу в качестве аргументов значения указываются в круглых скобках после имени метода.

Обычно поля и методы описываются с ключевым словом, определяющим уровень доступа. Если использовать ключевое слово `public`, то соответствующее поле или метод будут открытыми. К открытым полям и методам можно обращаться не только в программном коде класса, но и вне класса. Если ключевое слово `public` не указать, то поле или метод будут закрытыми. К закрытым полям и методам доступ имеется только внутри класса.

НА ЗАМЕТКУ

Кроме ключевого слова `public` можно использовать ключевые слова `private` и `protected`. С ключевым словом `private` описывают закрытые члены класса. Другими словами, если нужно сделать поле или метод закрытыми, то это поле или метод описываются с ключевым словом `private` или вообще без ключевого слова, определяющего уровень доступа. Ключевое слово `protected` используют при описании защищенных членов класса. Защищенные члены класса обсуждаются в рамках темы, связанной с наследованием.

Например, ниже приведено описание класса с одним целочисленным полем.

```
class MyClass{
   // Целочисленное поле:
   public int number;
}
```

Класс называется `MyClass`, и содержит он всего одно поле, которое называется `number`, является целочисленным (тип `int` — то есть

значением поля может быть целое число) и открытым (описано с ключевым словом `public`).

А вот еще один пример описания класса:

```
class MyClass{
   // Целочисленное поле:
   public int number;
   // Символьное поле:
   public char symbol;
   // Метод:
   public void show(){
      // Отображение значений полей:
      Console.WriteLine("Целое число: "+number);
      Console.WriteLine("Символ: "+symbol);
   }
}
```

У этой версии класса `MyClass` два поля (целочисленное `number` и символьное `symbol`). А еще в классе описан метод `show()`. Метод не возвращает результат (описан с ключевым словом `void`), и у него нет аргументов. При вызове метода выполняются две команды, которыми в консольное окно выводятся значения полей `number` и `symbol`. И вот здесь есть два важных момента. Первый момент связан с тем, что описание метода в классе еще не означает, что код этого метода выполняется. Чтобы код метода выполнялся, метод следует вызвать. А для вызова метода нужен объект, поскольку, если метод не статический, то вызывается он из объекта. Следовательно, необходимо создать объект и вызвать из него метод.

ПОДРОБНОСТИ

Необходимость вызывать метод из объекта поясним еще на одном отвлеченном примере. Допустим, имеются три кота одной породы: Мурчик, Барсик и Рыжик. Их мы можем отождествлять с объектами. Это объекты некоторого типа или класса, который можно было бы назвать Кот. У каждого из трех котов есть метод, который условно можно назвать «ловить мышей». Если мы хотим поймать реальную мышь, то ловить ее должен совершенно конкретный кот — то есть Мурчик, Барсик или Рыжик. Никакой абстрактный Кот мышей ловить не будет.

Второй момент связан с тем, что подразумевается под идентификаторами number и symbol в описании метода show() в классе MyClass. Мы уже знаем, что речь идет о полях. Но о полях какого объекта? Ответ, наверное, очевиден: имеются в виду поля объекта, из которого вызывается метод. Другими словами, если из некоторого объекта, созданного на основе класса MyClass, будет вызван метод show(), то этим методом в консольном окне будут отображены значения полей number и symbol объекта, из которого вызван метод.

Нам осталось выяснить, как создается объект класса. Общая схема похожа на ту, что используется при реализации массивов. Нам понадобится собственно сам объект, а также переменная, через которую мы будем обращаться к объекту. Такая переменная называется объектной переменной, и ее значением является ссылка на объект.

Объектная переменная объявляется так же просто, как и переменная базового типа, но только в качестве идентификатора типа указывается имя класса. Например, для объявления объектной переменной с именем obj, которая могла бы ссылаться на объект класса MyClass, используем инструкцию MyClass obj. Но объявление объектной переменной не приводит к созданию объекта. Объектная переменная может лишь ссылаться на объект. Сам объект нужно создать. Создается объект с помощью оператора new. После оператора new указывается имя класса с пустыми круглыми скобками. Так, для создания объекта класса MyClass используем инструкцию new MyClass(). При выполнении этой инструкции создается объект. Инструкция также возвращает результат — ссылку на созданный объект. Эту ссылку можно присвоить значением объектной переменной (в нашем случае это объектная переменная obj). Поэтому процесс создания объекта класса MyClass может быть описан командами:

```
MyClass obj;
obj=new MyClass();
```

Эти две команды можно объединить в одну, которой объявляется объектная переменная и сразу ей значением присваивается ссылка на вновь созданный объект:

```
MyClass obj=new MyClass();
```

Понятно, что тип объектной переменной (класс, указанный при объявлении переменной) должен совпадать с типом объекта (классом, на основе которого создан объект), ссылка на который присваивается

переменной. На рис. 6.1 проиллюстрирован принцип реализации объектов в языке C#.

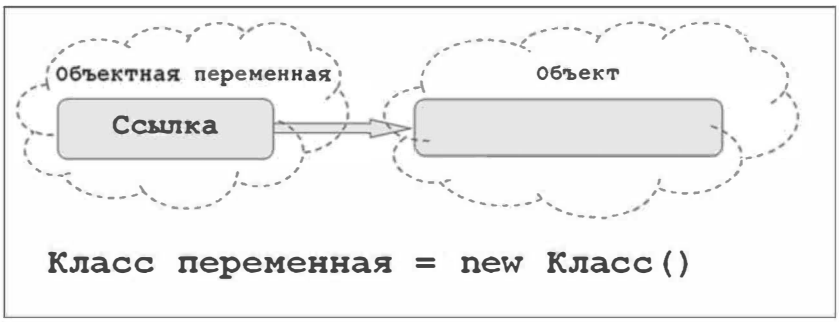


Рис. 6.1. Реализация объектов в C#

НА ЗАМЕТКУ

Несложно заметить, что способ реализации объектов напоминает способ реализации массивов: там тоже есть переменная (переменная массива), которая ссылается собственно на массив, а массив создается с помощью оператора `new`.

ПОДРОБНОСТИ

При создании объекта класса после инструкции `new` указывается не просто имя класса с круглыми скобками — это вызывается конструктор класса. Конструкторы обсуждаются немного позже. Пока же заметим, что скобки после имени класса могут быть и не пустыми. В общем случае они предназначены для передачи аргументов конструктору, вызываемому при создании объекта.

При создании объекта под все его поля выделяется место в памяти. С этими полями можно работать как с обычными переменными. Но нужно помнить, что у каждого объекта поля свои. Поэтому нет смысла обращаться к полю, не указав, к какому объекту оно относится.

НА ЗАМЕТКУ

Поля, как и методы, бывают статические и обычные (не статические). Статическое поле существует безотносительно к наличию или отсутствию объектов класса. Оно «не привязано» ни к какому конкретному объекту. Мы же обсуждаем обычные поля. Обычное поле существует только в контексте объекта. Нет объекта — нет поля.

При обращении к полю объекта используется «точечный синтаксис»: указывается имя объекта, точка и после нее — название поля. Общий формат ссылки выглядит как `объект.поле`. Например, если в классе `MyClass` описано поле с названием `number` и на основании этого класса создается объект `obj` (объектная переменная `obj` ссылается на созданный объект), то для обращения к полю `number` объекта `obj` используем инструкцию `obj.number`.

Методы (не статические) вызываются также с использованием «точечного синтаксиса». Для вызова метода из объекта необходимо указать имя объекта и через точку название метода (с аргументами, передаваемыми методу — если они нужны). Выглядит это как `объект.метод(аргументы)`. Например, если мы хотим вызвать метод `show()` из объекта `obj`, то команда вызова этого метода такая: `obj.show()`. Здесь предполагается, что методу `show()` при вызове аргументы не передаются (метод без аргументов). Метод `show()` при этом имеет автоматический доступ к полям и прочим методам объекта `obj`. Далее мы на конкретных примерах рассмотрим, как в программе описываются классы, создаются объекты, вызываются методы из объектов и выполняется обращение к полям объектов.

Использование объектов

> Я могу пронизать пространство и уйти в прошлое.
>
> *из к/ф «Иван Васильевич меняет профессию»*

Далее мы все, что было описано выше, используем на практике. В программе описывается класс `MyClass`, в котором есть два поля (целочисленное `number` и символьное `symbol`), а также метод `show()`, которым отображаются значения полей. В главном методе программы на основе класса `MyClass` создается два объекта (`A` и `B`), полям объектов присваиваются значения, а с помощью метода `show()` значения полей объектов выводятся в консольное окно. Соответствующий программный код приведен в листинге 6.1.

Листинг 6.1. Описание класса и создание объектов

```
using System;
// Описание класса:
```

```
class MyClass{
   // Целочисленное поле:
   public int number;
   // Символьное поле:
   public char symbol;
   // Метод:
   public void show(){
      // Отображение значения целочисленного поля:
      Console.WriteLine("Целочисленное поле: "+number);
      // Отображение значения символьного поля:
      Console.WriteLine("Символьное поле: "+symbol);
   }
}
// Класс с главным методом:
class UsingObjsDemo{
   // Главный метод:
   static void Main(){
      // Первый объект:
      MyClass A=new MyClass();
      // Объектная переменная:
      MyClass B;
      // Второй объект:
      B=new MyClass();
      // Присваивание значений полям первого объекта:
      A.number=123;
      A.symbol='A';
      // Присваивание значений полям второго объекта:
      B.number=321;
      B.symbol='B';
      // Вызов методов:
      Console.WriteLine("Первый объект");
      A.show();
```

```
      Console.WriteLine("Второй объект");
      B.show();
   }
}
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 6.1)

```
Первый объект
Целочисленное поле: 123
Символьное поле: A
Второй объект
Целочисленное поле: 321
Символьное поле: B
```

В программе, кроме класса с главным методом программы, описан еще и класс `MyClass`. В методе `Main()` создаются объекты этого класса. Их всего два. Первый объект создается командой `MyClass A=new MyClass()`. В данном случае объявляется объектная переменная `A` класса `MyClass` и ей значением присваивается ссылка на объект, который создается инструкцией `new MyClass()`. Со вторым объектом ситуация немного иная. Сначала командой `MyClass B` объявляется объектная переменная `B` класса `MyClass`. Создание объекта и присваивание ссылки на этот объект переменной `B` выполняется отдельной командой `B=new MyClass()`.

После того как объекты `A` и `B` созданы, их полям присваиваются значения. Значения полям объекта `A` присваиваем командами `A.number=123` и `A.symbol='A'`. Полям объекта `B` значения присваиваются с помощью команд `B.number=321` и `B.symbol='B'`. Для проверки значений полей объектов `A` и `B` используем соответственно команды `A.show()` и `B.show()`. Этими командами из объектов `A` и `B` вызывается метод `show()`. В классе `MyClass` метод описан таким образом, что им в консольное окно выводятся значения полей `number` и `symbol`. Объект, к которому относятся эти поля, в описании метода в классе `MyClass` явно не указан. В таком случае подразумеваются поля того объекта, из которого вызывается метод. Это и происходит на практике: при вызове метода `show()` из объекта `A` отображаются значения полей объекта `A`, а при вызове метода `show()` из объекта `B` отображаются значения поле объекта `B`.

НА ЗАМЕТКУ

В главном методе программы обращение к полям и методам объектов выполняется в виде `A.number`, `B.symbol`, `A.show()` и так далее. Это возможно, поскольку поля `number` и `symbol`, а также метод `show()` в классе `MyClass` описаны с ключевым словом `public` — то есть поля и методы являются открытыми. Если бы они не были открытыми, то в главном методе программы мы не смогли бы напрямую обратиться к полям и методу объектов, созданных на основе класса `MyClass`.

Еще один небольшой пример, связанный с использованием классов и объектов, представлен в листинге 6.2. Во многом он напоминает предыдущую программу из листинга 6.1, но все же в нем есть и существенные отличия.

Листинг 6.2. Присваивание объектов

```
using System;
// Описание класса:
class MyClass{
   // Целочисленное поле:
   public int number;
   // Метод для отображения значения поля:
   public void show(){
      Console.WriteLine("Значение поля: "+number);
   }
}
// Класс с главным методом:
class AnotherObjsDemo{
   // Главный метод:
   static void Main(){
      // Объектные переменные:
      MyClass A, B;
      // Создание объекта:
      A=new MyClass();
      // Присваивание объектных переменных:
```

```
        B=A;
        // Присваивание значения полю через первую
        // объектную переменную:
        A.number=123;
        // Вызов метода через вторую объектную переменную:
        B.show();
        // Присваивание значения полю через вторую
        // объектную переменную:
        B.number=321;
        // Вызов метода через первую объектную переменную:
        A.show();
    }
}
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 6.2)

```
Значение поля: 123
Значение поля: 321
```

В данной программе мы немного упростили класс `MyClass`: теперь у класса всего одно целочисленное поле `number`. Метод `show()` при вызове отображает в консольном окне значение поля `number` объекта, из которого он вызывается.

В главном методе программы мы объявляем две объектные переменные `A` и `B` класса `MyClass` (команда `MyClass A, B`). Но объект создается только один: командой `A=new MyClass()` ссылка на созданный объект записывается в переменную `A`. Затем командой `B=A` переменной `B` присваивается значение переменной `A`. Это важный момент: мы новый объект не создаем. Переменной `B` присваивается переменная `A`, которая уже ссылается на ранее созданный объект. Что же происходит в этом случае? Чтобы получить ответ на вопрос, следует учесть, что фактическим значением переменной `A` является не объект, а ссылка на объект. Мы можем эту ситуацию интерпретировать так, как если бы в переменную `A` был записан некий адрес объекта, на который ссылается переменная. Если так, то при выполнении команды `B=A` в переменную `B`

записывается адрес, который записан в переменную A. В итоге и переменная A, и переменная B содержат адрес одного и того же объекта. Это означает, что обе переменные ссылаются на один и тот же объект. Поэтому когда командой A.number=123 мы присваиваем значение полю number через переменную A, а затем вызываем метод show() через переменную B (команда B.show()), то в обоих случаях речь идет об одном и том же объекте. Аналогично, если изменить значение поля number с помощью команды B.number=321, то при вызове метода show() через переменную A (команда A.show()) в консольном окне отобразится новое значение поля.

Закрытые члены класса и перегрузка методов

Чего он меня все нугает? Что меня пугать?
У меня три ножизненных заключения.

из к/ф «Формула любви»

Ранее мы познакомились с перегрузкой статических методов. Но перегружать можно и обычные методы. То есть в классе может быть описано несколько версий одного и того же метода. У таких версий одинаковые названия, но разные аргументы. При вызове метода версия метода определяется по количеству и типу аргументов, фактически переданных методу. Перегрузка методов проиллюстрирована в программе в листинге 6.3. Там же используются закрытые члены класса (поля класса описаны с ключевым словом private). Рассмотрим представленный ниже программный код.

Листинг 6.3. Закрытые члены класса и перегрузка методов

```
using System;
// Описание класса:
class MyClass{
   // Закрытое целочисленное поле:
   private int number;
   // Закрытое символьное поле:
   private char symbol;
   // Открытый метод для отображения значения полей:
```

```
   public void show(){
      Console.WriteLine("Поля объекта: "+number+" и "+symbol);
   }
   // Открытый метод для присваивания значений полям.
   // Версия с двумя аргументами:
   public void set(int n, char s){
      number=n; // Значение целочисленного поля
      symbol=s; // Значение символьного поля
   }
   // Открытый метод для присваивания значений полям.
   // Версия с одним целочисленным аргументом:
   public void set(int n){
      number=n;   // Значение целочисленного поля
      symbol='B'; // Значение символьного поля
   }
   // Открытый метод для присваивания значений полям.
   // Версия без аргументов:
   public void set(){
      // Вызов версии метода с двумя аргументами:
      set(100,'A');
   }
}
// Главный класс:
class MethodsDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта:
      MyClass obj=new MyClass();
      // Присваивание значений полям:
      obj.set();
      // Отображение значений полей:
      obj.show();
```

```
        // Присваивание значений полям:
        obj.set(200);
        // Отображение значений полей:
        obj.show();
        // Присваивание значений полям:
        obj.set(300,'C');
        // Отображение значений полей:
        obj.show();
    }
}
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 6.3)

```
Поля объекта: 100 и A
Поля объекта: 200 и B
Поля объекта: 300 и C
```

Как отмечалось выше, в данном примере при описании класса `MyClass` поля `number` и `symbol` описаны с ключевым словом `private`. Это означает, что поля закрытые. К ним можно обращаться в программном коде класса, но за пределами класса они недоступны для прямого обращения.

ПОДРОБНОСТИ

Мы не можем обратиться к закрытым членам класса вне пределов класса напрямую. Допустим, имеется объект `obj` класса `MyClass`. Тогда в главном методе программы обратиться к полям `number` и `symbol` (при условии, что они закрытые) через имя объекта не получится. Зато мы можем вызвать открытый метод, который обращается к этим полям. Такой подход использован в рассматриваемой программе.

Как и ранее, в классе присутствует метод `show()`, отображающий в консольном окне значения полей `number` и `symbol`. Метод открытый, поэтому мы сможем его вызывать в главном методе программы из объектов класса `MyClass` (в реальности там всего один объект `obj`). Кроме метода `show()`, в классе `MyClass` описаны три версии метода `set()`. Метод `set()` предназначен для присваивания значений полям объекта. Дело в том, что поскольку поля закрытые, то после создания

объекта напрямую (прямым обращением к полям) значения им присвоить не удастся. Но можно вызвать открытый метод, который и присвоит полям значения. Таким методом является `set()`. Если он вызывается с двумя аргументами (целочисленным и символьным), то значения аргументов присваиваются полям объекта, из которого вызывается метод. Если метод вызывается с одним целочисленным аргументом, то значение аргумента присваивается целочисленному полю `number`, а полю `symbol` присваивается значение `'B'`. Наконец, в классе `MyClass` описана версия метода `set()` без аргументов. В описании этой версии метода есть всего одна команда `set(100,'A')`, которой вызывается версия метода с двумя аргументами. В результате поле `number` получает значение `100`, а поле `symbol` получает значение `'A'`.

НА ЗАМЕТКУ

О рекурсии здесь речь не идет, поскольку одна версия метода вызывает другую (при рекурсии внутри метода вызывается та же версия метода).

В главном методе программы командой `MyClass obj=new MyClass()` создается объект `obj` класса `MyClass`. Присваивание значений полям объекта `obj` выполняется с помощью команды `obj.set()`. В результате поле `number` объекта `obj` получает значение `100`, а поле `symbol` этого объекта получает значение `'A'`. Проверяем значения полей объекта с помощью команды `obj.show()`. После выполнения команды `obj.set(200)` поле `number` объекта `obj` получает значение `200`, а поле `symbol` получает значение `'B'`. Наконец, после выполнения команды `obj.set(300,'C')` значение поля `number` объекта `obj` равно `300`, а значение поля `symbol` объекта `obj` равно `'C'`.

НА ЗАМЕТКУ

Поля `number` и `symbol` закрытые, поэтому доступа из программного кода вне пределов класса к этим полям нет. Но методы `set()` и `show()` описаны в классе, поэтому они имеют доступ к полям. В главном методе программы мы не можем напрямую обратиться к закрытым полям, но можем вызвать открытые методы, которые имеют доступ к полям. Таким образом, получается, что мы «спрятали» поля и ввели «посредников» (методы), которые позволяют нам выполнять операции с полями — но только те, которые определены в методах. Это очень удобный подход, особенно если необходимо ограничить или упорядочить режим доступа к полям (или другим членам).

Конструктор

Вот что крест животворящий делает!

из к/ф «Иван Васильевич меняет профессию»

В рассмотренных ранее примерах после создания объекта нам приходилось напрямую или с помощью специальных методов присваивать значения полям объекта. Это не всегда удобно. Хорошо было бы иметь возможность задавать значения полей объекта уже на этапе создания объекта. И такая возможность имеется. Связана она с использованием *конструктора*.

Конструктор — это специальный метод, который автоматически вызывается при создании объекта. Поскольку это метод специальный, то и описывается он специальным способом. Основные правила описания конструктора в классе такие:

- Имя конструктора совпадает с именем класса. Обычно описывается с ключевым словом `public`.
- Конструктор не возвращает результат, и идентификатор типа результата для конструктора не указывается (в отличие от обычных методов, которые в подобном случае описываются с идентификатором `void`).
- У конструктора могут быть аргументы. Аргументы передаются конструктору при создании объекта.
- Конструктор можно перегружать. Это означает, что в классе может быть описано несколько конструкторов. Каждая версия конструктора отличается типом и/или количеством аргументов.

Если в инструкции создания объекта на основе оператора `new` после имени класса в круглых скобках указываются определенные значения, то это и есть аргументы, которые передаются конструктору.

ПОДРОБНОСТИ

Инструкция вида `new Класс()` представляет собой команду вызова конструктора `Класса`, причем конструктор вызывается без аргументов. Команда вида `new Класс(аргументы)` означает, что при создании объекта вызывается конструктор `Класса` и конструктору передаются `аргументы`.

Как отмечалось выше, в классе может быть описано несколько конструкторов (или несколько версий конструктора). В таком случае версия конструктора, вызываемая при создании объекта, определяется по количеству и типу аргументов, переданных конструктору в команде создания объекта. Здесь имеет место полная аналогия с перегрузкой методов. Небольшой пример использования конструкторов представлен в листинге 6.4.

Листинг 6.4. Использование конструктора

```
using System;
// Описание класса с конструктором:
class MyClass{
   // Закрытые поля:
   public int num;      // Целочисленное поле
   public char symb;    // Символьное поле
   public string txt;   // Текстовое поле
   // Открытый метод для отображения значений полей:
   public void show(){
      Console.WriteLine("Поля: {0}, \'{1}\' и \"{2}\"", num, symb, txt);
   }
   // Конструктор без аргументов:
   public MyClass(){
      // Значения полей:
      num=100;
      symb='A';
      txt="Красный";
   }
   // Конструктор с одним целочисленным аргументом:
   public MyClass(int n){
      // Значения полей:
      num=n;
      symb='B';
      txt="Желтый";
   }
```

```
   // Конструктор с двумя аргументами:
   public MyClass(int n, char s){
      // Значения полей:
      num=n;
      symb=s;
      txt="Зеленый";
   }
   // Конструктор с тремя аргументами:
   public MyClass(int n, char s, string t){
      // Значения полей:
      num=n;
      symb=s;
      txt=t;
   }
   // Конструктор с одним текстовым аргументом:
   public MyClass(string t){
      // Значения полей:
      num=0;
      symb='Z';
      txt=t;
   }
}
// Класс с главным методом:
class ConstructorsDemo{
   // Главный метод:
   static void Main(){
      // Создание объектов.
      // Вызывается конструктор без аргументов:
      MyClass A=new MyClass();
      // Проверяются значения полей объекта:
      A.show();
      // Вызывается конструктор с целочисленным аргументом:
```

```
        MyClass B=new MyClass(200);
        // Проверяются значения полей объекта:
        B.show();
        // Вызывается конструктор с двумя аргументами:
        MyClass C=new MyClass(300,'C');
        // Проверяются значения полей объекта:
        C.show();
        // Вызывается конструктор с тремя аргументами:
        MyClass D=new MyClass(400,'D',"Синий");
        // Проверяются значения полей объекта:
        D.show();
        // Вызывается конструктор с символьным аргументом:
        MyClass F=new MyClass('A');
        // Проверяются значения полей объекта:
        F.show();
        // Вызывается конструктор с текстовым аргументом:
        MyClass G=new MyClass("Серый");
        // Проверяются значения полей объекта:
        G.show();
    }
}
```

Как выглядит результат выполнения программы, показано ниже.

Результат выполнения программы (из листинга 6.4)

```
Поля: 100, 'A' и "Красный"
Поля: 200, 'B' и "Желтый"
Поля: 300, 'C' и "Зеленый"
Поля: 400, 'D' и "Синий"
Поля: 65, 'B' и "Желтый"
Поля: 0, 'Z' и "Серый"
```

Мы в данном случае имеем дело с классом `MyClass`, у которого есть три закрытых поля: целочисленное, символьное и текстовое. В классе

описан открытый метод `show()`, отображающий в консольном окне значения полей.

ПОДРОБНОСТИ

В теле метода `show()` использована команда `Console.WriteLine ("Поля: {0}, \'{1}\' и \"{2}\"", num, symb, txt)`. Командой в консольном окне отображается строка `"Поля: {0}, \'{1}\' и \"{2}\""`, в которой вместо блока `{0}` отображается значение целочисленного поля `num`, вместо блока `{1}` отображается значение символьного поля `symb`, а вместо блока `{2}` отображается значение текстового поля `txt`. Также в отображаемой текстовой строке мы используем одинарные и двойные кавычки. Одинарные кавычки добавляем в текстовую строку с помощью инструкции `\'`, а двойные кавычки добавляются в текстовую строку с помощью инструкции `\"`.

Кроме метода `show()`, в классе описано пять версий конструктора: без аргументов, с одним целочисленным аргументом, с одним текстовым аргументом, с двумя аргументами (целочисленным и символьным) и с тремя аргументами (целочисленным, символьным и текстовым). Каждая из версий конструктора описана с ключевым словом `public`.

НА ЗАМЕТКУ

Конструктор — это метод. Если мы хотим вызывать конструктор (то есть создавать объекты) вне пределов класса (а мы хотим), то конструктор должен быть открытым. Чтобы он был открытым, конструктор описывают с ключевым словом `public` — как и в случае, когда в классе описывается открытый метод.

Название конструктора совпадает с названием класса: в данном случае класс называется `MyClass`, отсюда и название конструктора. В теле конструктора размещаются команды, которые выполняются при создании объекта. Например, так описана версия конструктора без аргументов (комментарии удалены):

```
public MyClass(){
   num=100;
   symb='A';
   txt="Красный";
}
```

Это означает, что если мы создаем объект и конструктору не передаем аргументы (как это имеет место в команде `MyClass A=new MyClass()` в главном методе программы), то целочисленному полю создаваемого объекта будет присвоено значение `100`, символьное поле получит значение `'A'`, а текстовое поле получит значение `"Красный"`.

Конструктор с целочисленным аргументом описан таким образом, что его аргумент определяет значение целочисленного поля создаваемого объекта, а символьное и текстовое поле получают соответственно значения `'B'` и `"Желтый"`.

Конструктору могут передаваться два аргумента: первый целочисленный и второй символьный. Они определяют значения целочисленного и символьного полей объекта. Текстовое поле получает значение `"Зеленый"`. Если же конструктору передаются три аргумента (целое число, символ и текст), то они определяют значения полей объекта, который создается.

Кроме версии конструктора с одним целочисленным аргументом, класс `MyClass` содержит описание версии конструктора с одним текстовым аргументом. Текстовое значение, переданное аргументом конструктору, задает значение текстового поля создаваемого объекта. При этом целочисленное поле получает значение `0`, а символьному полю присваивается значение `'Z'`.

В главном методе программы создается несколько объектов. Каждый раз используются разные конструкторы. Значения полей объектов проверяем, вызывая из объектов метод `show()`. Например, при создании объекта командой `MyClass B=new MyClass(200)` вызывается конструктор с одним целочисленным аргументом. Команда `MyClass C=new MyClass(300,'C')` подразумевает создание объекта с использованием конструктора с двумя аргументами. Конструктор с тремя аргументами вызывается, когда объект создается командой `MyClass D=new MyClass(400,'D',"Синий")`.

В команде `MyClass G=new MyClass("Серый")` конструктору передается один текстовый аргумент. Такая версия конструктора в классе `MyClass` описана и будет вызвана при создании объекта. Но вот в команде `MyClass F=new MyClass('A')` конструктору передается символьный аргумент, и такой версии конструктора (с одним символьным аргументом) в классе `MyClass` нет. Зато в классе `MyClass` есть версия конструктора с одним целочисленным аргументом, а в языке C#

разрешено автоматическое преобразование символьного типа в целочисленный. В результате при создании объекта командой `MyClass F=new MyClass('A')` вызывается конструктор с целочисленным аргументом, а символьное значение `'A'` фактически переданного аргумента преобразуется к числовому значению `65` (код символа `'A'` в кодовой таблице).

ПОДРОБНОСТИ

Если в классе не описан ни один конструктор, то при создании объекта используется *конструктор по умолчанию*: у такого конструктора нет аргументов, и никакие дополнительные действия с объектом он не выполняет. Но если в классе описана хотя бы одна версия конструктора, то конструктор по умолчанию больше не доступен. Так, если в рассмотренном выше примере в классе `MyClass` не описать версию конструктора без аргументов, то команда `MyClass A=new MyClass()` станет некорректной.

Деструктор

Замуровали, демоны!

из к/ф «Иван Васильевич меняет профессию»

Если конструктор автоматически вызывается при создании объекта, то *деструктор* — это метод, который автоматически вызывается при удалении объекта из памяти. При описании деструктора нужно придерживаться следующих правил.

- Имя деструктора начинается с тильды `~`, после которой указывается название класса. Если мы описываем деструктор для класса `MyClass`, то называться такой деструктор будет `~MyClass`.
- Деструктор не возвращает результат, идентификатор типа результата для него не указывается, также не указывается ключевое слово `public`.
- У деструктора нет аргументов, и он не может быть перегружен (в классе только один деструктор).

Перед тем как объект удаляется из памяти, вызывается деструктор. Поэтому в теле деструктора размещаются команды, которые необходимо выполнить перед самым удалением объекта.

НА ЗАМЕТКУ

Как отмечалось выше, деструктор автоматически вызывается при удалении объекта из памяти. Но проблема в том, что выход объекта из области видимости (потеря ссылки на объект) не означает, что объект будет сразу же удален. Он будет удален, но это может произойти через некоторое время. За удаление «ненужных» объектов отвечает система «сборки мусора». Она работает автономно и удаляет те объекты, на которые в программе нет ссылок. Поэтому, если на объект теряется последняя ссылка, он автоматически становится кандидатом на удаление. Когда именно он будет удален, «решает» «сборщик мусора», а не программа. Деструктор вызывается при удалении объекта. Получается, что точное время вызова деструктора неизвестно. Мы лишь знаем, что он будет вызван. Эту особенность нужно учитывать при использовании деструкторов в классах.

Иллюстрация к использованию деструктора в программе представлена в листинге 6.5.

Листинг 6.5. Использование деструктора

```
using System;
// Класс с конструктором и деструктором:
class MyClass{
   // Закрытое текстовое поле:
   private string name;
   // Конструктор:
   public MyClass(string txt){
      // Присваивание значения полю:
      name=txt;
      // Отображение сообщения:
      Console.WriteLine("Создан объект \"{0}\"", name);
   }
   // Деструктор:
   ~MyClass(){
      // Отображение сообщения:
      Console.WriteLine("Удален объект \"{0}\"", name);
   }
```

```
}
// Класс с главным методом:
class DestructorDemo{
   // Статический метод:
   static void maker(string txt){
      // Создание анонимного объекта:
      new MyClass(txt);
   }
   // Главный метод:
   static void Main(){
      // Создание объекта:
      MyClass A=new MyClass("Первый");
      // Создание анонимного объекта:
      new MyClass("Второй");
      // Новый объект:
      A=new MyClass("Третий");
      // Вызов статического метода:
      maker("Четвертый");
      // Новый объект:
      A=new MyClass("Пятый");
   }
}
```

Результат выполнения программы, скорее всего, будет таким.

Результат выполнения программы (из листинга 6.5)

```
Создан объект "Первый"
Создан объект "Второй"
Создан объект "Третий"
Создан объект "Четвертый"
Создан объект "Пятый"
Удален объект "Пятый"
Удален объект "Первый"
```

```
Удален объект "Четвертый"
Удален объект "Третий"
Удален объект "Второй"
```

Мы описываем класс `MyClass`, и в этом классе есть закрытое текстовое поле `name`. Конструктор у класса всего один. Аргументом конструктору передается текстовое значение, которое присваивается полю `name`. После присваивания значения полю `name` командой `Console.WriteLine("Создан объект \"{0}\"", name)` в консольное окно выводится сообщение о том, что создан новый объект. Сообщение содержит значение поля `name` созданного объекта.

НА ЗАМЕТКУ

Один факт создания объекта означает, что в консольном окне появляется сообщение со значением закрытого текстового поля этого объекта. Проще говоря, когда с использованием оператора `new` создается объект класса `MyClass`, в консольном окне появляется сообщение. Все это — благодаря конструктору, в котором есть соответствующая команда.

Также стоит заметить, что поскольку в классе `MyClass` описан только один конструктор (с текстовым аргументом), то при создании объекта класса необходимо передать конструктору текстовый аргумент. Других способов создания объекта класса нет.

Еще в классе есть деструктор. В теле деструктора всего одна команда `Console.WriteLine("Удален объект \"{0}\"", name)`, которой в консольное окно выводится сообщение об удалении объекта и значении поля `name` этого объекта.

НА ЗАМЕТКУ

Каждый раз при удалении объекта из памяти в консольном окне появляется сообщение со значением закрытого текстового поля объекта, который удаляется. Это происходит благодаря наличию деструктора в классе `MyClass`.

В главном классе программы кроме метода `Main()` описан еще и статический метод `maker()`. У метода есть текстовый аргумент (обозначен как `txt`). В теле метода всего одна команда `new MyClass(txt)`, которой создается объект класса `MyClass`. Пикантность ситуации в том,

что ссылка на созданный объект не записывается в объектную переменную. Получается, что объект создается, но нет переменной, через которую можно было бы получить доступ к этому объекту. Такие объекты обычно называют *анонимными*.

НА ЗАМЕТКУ

Анонимные объекты используются в тех случаях, когда нужен «одноразовый» объект — то есть объект, который используется один раз, после чего больше не нужен. В таком случае инструкция, создающая объект, не присваивается объектной переменной, а размещается в том месте в выражении, где должна быть ссылка на объект.

В данном случае мы не планируем каким-либо образом использовать созданный объект, поэтому нет необходимости записывать в отдельную переменную ссылку на него.

В главном методе программы командой `MyClass A=new MyClass("Первый")` создается первый объект. Вследствие вызова конструктора при выполнении этой команды в консольном окне появляется сообщение `Создан объект "Первый"`. После этого командой `new MyClass("Второй")` создается второй объект, и в консольном окне появляется сообщение `Создан объект "Второй"`. Созданный объект анонимный, поскольку ссылка на него в объектную переменную не записывается. Раз так, то получается, что программа не содержит переменных со ссылками на созданный объект, и он становится кандидатом на удаление. Начиная с этого момента, в консольном окне потенциально может появиться сообщение об удалении объекта (но в данном случае пока не появляется).

Командой `A=new MyClass("Третий")` создается третий объект (при создании объекта в консоли появляется сообщение `Создан объект "Третий"`), и ссылка на него записывается в переменную `A`. Раньше в эту переменную была записана ссылка на первый объект. Теперь переменная ссылается на третий объект. Следовательно, на первый объект ссылок в программе не осталось. Поэтому первый объект также становится кандидатом на удаление и может быть удален системой «сборки мусора».

При вызове статического метода `maker()` с аргументом `"Четвертый"` создается анонимный объект. При его создании появляется сообщение `Создан объект "Четвертый"`. Поскольку объект анонимный, то он сразу после создания попадает в категорию подлежащих удалению.

Наконец, последней командой `A=new MyClass("Пятый")` в главном методе создается еще один (пятый по счету) объект (в консоли появляется сообщение `Создан объект "Пятый"`), и ссылка на него записывается в переменную `A`. До этого переменная `A` ссылалась на третий объект. Следовательно, третий объект подлежит удалению, поскольку на него в программе ссылок нет.

На последний, пятый объект, в программе ссылка есть (записана в переменную `A`). Но программа завершает работу. А при завершении выполнения программы из памяти удаляются все переменные и объекты. Поэтому пятый объект также должен быть удален. В итоге мы получаем, что все пять созданных объектов класса `MyClass` должны быть удалены. И они начинают удаляться. Как видно из результата выполнения программы, первым удаляется объект `"Пятый"`. Затем удаляется объект `"Первый"`, затем `"Четвертый"`, `"Третий"` и, наконец, `"Второй"`. Здесь еще раз подчеркнем, что удалением объектов из памяти занимается система «сборки мусора», поэтому нельзя однозначно предугадать, когда именно объекты будут удалены. Можно быть уверенным лишь в том, что это произойдет.

Статические члены класса

Приедут тут всякие: без профессии, без подушек...

из к/ф «Девчата»

Поля и методы могут быть *статическими*. Особенность статических полей и методов в том, что они могут использоваться без создания объектов.

> **НА ЗАМЕТКУ**
>
> Со статическими методами мы уже встречались ранее и обсуждали их. Но там речь шла о статических методах, описанных в том же классе, что и главный метод программы. Соответственно, мы использовали статические методы в том же классе, в котором они были описаны. Но в общем случае статический метод может быть описан в одном классе, а использован в другом.

Статические члены класса описываются с ключевым словом `static`. Для обращения к статическому члену класса (полю или методу) указывается имя класса, после которого через точку указывают имя поля или имя метода (с аргументами или без). Таким образом, если при обращении

к обычным полям и методам нам нужно указать объект, то при обращении к статическим полям и методам вместо объекта указывается класс. Если статическое поле или метод используются в том же классе, где они описаны, то имя класса при обращении к полю или методу можно не указывать.

НА ЗАМЕТКУ

Статические поля играют роль глобальных переменных. Они «не привязаны» к какому-то конкретному объекту и существуют «сами по себе». Единственное ограничение — статические поля нужно описывать в классе и при обращении указывать имя этого класса.
Аналогично, статические методы в языке C# играют примерно ту же роль, что и функции в языке C++. Статический метод — это, по сути, блок команд, которые не подразумевают использования объекта, и поэтому для выполнения этих команд (вызова метода) объект как таково не нужен.

ПОДРОБНОСТИ

При обращении к статическому полю или при вызове статического метода вместо имени класса можно указать и имя объекта этого класса, причем объект может быть любым (главное, чтобы того класса, в котором описан статический член). Во всех случаях речь идет о статических членах. Но обращаться к статическому члену через объект все же нежелательно, поскольку в этом случае создается иллюзия, что соответствующее поле или метод являются обычными (не статическими), а это не так.

При работе со статическими методами существует очевидное ограничение: статический метод может обращаться только к статическим полям и методам в классе. Объяснение простое: поскольку статический метод существует без привязки к объекту, то нет смысла обращаться к полям и методам объекта (поскольку объекта как такового нет).

В листинге 6.6 представлена программа. В ней используется класс со статическим полем и методом.

Листинг 6.6. Знакомство со статическими полями и методами

```
using System;
// Класс со статическим полем и методом:
class MyClass{
   // Статическое поле:
```

```
   public static int code=100;
   // Статический метод:
   public static void show(){
      Console.WriteLine("Статическое поле: "+code);
   }
}
// Класс с главным методом:
class StaticDemo{
   // Главный метод:
   static void Main(){
      // Вызов статического метода:
      MyClass.show();
      // Обращение к статическому полю:
      MyClass.code=200;
      // Вызов статического метода:
      MyClass.show();
   }
}
```

Ниже представлен результат выполнения программы.

Результат выполнения программы (из листинга 6.6)

```
Статическое поле: 100
Статическое поле: 200
```

В программе описывается класс `MyClass`, у которого есть открытое статическое целочисленное поле `code` с начальным значением `100`, а также статический метод `show()` (без аргументов и не возвращающий результат). Метод при вызове отображает в консольном окне значение статического поля `code`.

В главном методе программы командой `MyClass.show()` вызывается статический метод `show()` класса `MyClass`. В результате в консоли отображается текущее значение статического поля `code`. Мы это значение можем изменить — например, с помощью команды `MyClass.code=200`, после чего значение статического поля становится равным `200`. Проверить это легко — достаточно вызвать метод

`show()` (команда `MyClass.show()`). В данном случае примечателен факт, что мы не создали ни одного объекта класса `MyClass`, но это не мешает нам использовать статические члены этого класса.

Еще один, на этот раз «математический» пример с использованием статических полей и методов представлен в листинге 6.7. В программе описан класс `MyMath`, в котором есть статические методы для вычисления значений синуса и экспоненты. В классе также имеется константное статическое поле, значение которого определяет иррациональное число π.

НА ЗАМЕТКУ

Что касается статических полей, то нередко они используются как статические константы: для такого поля указывается идентификатор типа (и если нужно — спецификатор уровня доступа), но вместо ключевого слова `static` указывается идентификатор `const`. Причина в том, что константные поля по умолчанию реализуются как статические. Значение константного поля указывается при объявлении и впоследствии не может быть изменено.

ПОДРОБНОСТИ

Для вычисления экспоненты используется следующее выражение: $\exp(x) \approx 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} \ldots + \frac{x^n}{n!} = \sum_{k=0}^{n} \frac{x^k}{k!}$. В программе описывается статический метод для вычисления суммы $\sum_{k=0}^{n} q_k = q_0 + q_1 + \ldots + q_n$, где при заданном значении аргумента x слагаемые в сумме вычисляются как $q_k = \frac{x^k}{k!}$. Сумма вычисляется с помощью оператора цикла, в котором после прибавления к сумме очередного слагаемого рассчитывается слагаемое для следующей итерации. При этом мы исходим из того, что если вычислена добавка q_k для текущей итерации, то добавка q_{k+1} для следующей итерации вычисляется как $q_{k+1} = q_k \times \frac{x}{k+1}$ (легко проверить, что если $q_k = \frac{x^k}{k!}$ и $q_k = \frac{x^{k+1}}{(k+1)!}$, то $\frac{q_{k+1}}{q_k} = \frac{x}{k+1}$).

Аналогичным образом вычисляется синус: используется формула $\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots + \frac{(-1)^n x^{2n+1}}{(2n+1)!} = \sum_{k=0}^{n} \frac{(-1)^k x^{2k+1}}{(2k+1)!}$. Подход используем тот же, что и при вычислении экспоненты — то есть вычисляется сумма вида $\sum_{k=0}^{n} q_k = q_0 + q_1 + \ldots + q_n$, но на этот раз $q_k = \frac{(-1)^k x^{2k+1}}{(2k+1)!}$, $q_{k+1} = \frac{(-1)^{k+1} x^{2k+3}}{(2k+3)!}$, $\frac{q_{k+1}}{q_k} = -\frac{x^2}{(2k+3)(2k+2)}$, и тогда $q_{k+1} = q_k \times \frac{(-1)x^2}{(2k+3)(2k+2)}$.

Для определения значения параметра *n* (верхняя граница в суммах, через которые вычисляется синус и экспонента) в программе (в соответствующем классе) объявлено статическое целочисленное поле. А значение π=3,141592 определяется через статическую константу.

Проанализируем представленный ниже программный код.

Листинг 6.7. Использование статических полей и методов

```
using System;
// Класс со статическими методами и полями:
class MyMath{
   // Константное поле (число "пи"):
   public const double Pi=3.141592;
   // Закрытое статическое поле (граница суммы):
   private static int N=100;
   // Статический метод для вычисления экспоненты:
   public static double exp(double x){
      // Сумма и добавка к сумме:
      double s=0, q=1;
      // Вычисление суммы:
      for(int k=0; k<=N; k++){
         s+=q;       // Прибавление добавки к сумме
         q*=x/(k+1); // Добавка для следующей итерации
      }
      // Результат:
      return s;
   }
   // Статический метод для вычисления синуса:
   public static double sin(double x){
      // Сумма и добавка к сумме:
      double s=0, q=x;
      // Вычисление суммы:
      for(int k=0; k<=N; k++){
         s+=q;       // Прибавление добавки к сумме
```

```
            // Добавка для следующей итерации:
            q*=(-1)*x*x/(2*k+2)/(2*k+3);
         }
         // Результат:
         return s;
      }
}
// Главный класс:
class StaticDemo{
   // Главный метод:
   static void Main(){
      // Аргумент для статических методов:
      double z=1;
      // Вычисление экспоненты:
      Console.WriteLine("exp({0})={1}", z, MyMath.exp(z));
      // Контрольное значение:
      Console.WriteLine("Контрольное значение: {0}", Math.Exp(z));
      // Новое значение аргумента:
      z=MyMath.Pi/4;
      // Вычисление синуса:
      Console.WriteLine("sin({0})={1}", z, MyMath.sin(z));
      // Контрольное значение:
      Console.WriteLine("Контрольное значение: {0}", Math.Sin(z));
   }
}
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 6.7)

```
exp(1)=2,71828182845905
Контрольное значение: 2,71828182845905
sin(0,785398)=0,707106665647094
Контрольное значение: 0,707106665647094
```

Мы описываем класс `MyMath`, в котором описаны статические методы `exp()` и `sin()` для вычисления соответственно экспоненты и синуса. Оба метода реализованы по общему принципу: объявляются две локальные числовые переменные `s` и `q`. В первую записывается значение для суммы, через которую вычисляется экспонента или синус, а во вторую заносится значение добавки к сумме. Начальное значение суммы в обоих случаях нулевое. При вычислении экспоненты первая добавка единичная, а для синуса первая добавка к сумме равна значению аргумента, переданного методу.

Сумма вычисляется с помощью оператора цикла, в котором за каждую итерацию к сумме прибавляется очередная добавка, а затем вычисляется добавка для следующей итерации. Количество слагаемых в сумме определяется значением статического целочисленного поля `N` (значение поля равно `100`). Поле описано со спецификатором уровня доступа `private`. Поэтому поле является закрытым и доступно только внутри кода класса `MyMath`.

Поле `Pi` описано с идентификатором `const`. Это означает, что поле константное (его значение изменить нельзя). Константные поля автоматически реализуются как статические. Поэтому в главном методе программы доступ к этому полю получаем с помощью инструкции `MyMath.Pi`. Там (в главном методе) вычисляются значения для экспоненты и синуса. Для сравнения приведены контрольные значения, которые вычисляются с помощью статических методов `Exp()` и `Sin()` класса `Math`. Несложно заметить, что совпадение более чем приемлемое.

НА ЗАМЕТКУ

Библиотечный класс `Math` (пространство имен `System`) содержит статические методы, через которые реализуются основные математические функции. Там также есть статические поля со значениями некоторых математических констант (основание натурального логарифма и число «пи»).

Ключевое слово this

— Владимир Николаевич, а может быть, мы все-таки на...
— Да, типичные марсиане.

из к/ф «Кин-дза-дза»

Мы уже знаем, что если в программном коде метода встречается название поля, то имеется в виду поле того объекта, из которого вызывается метод. Концептуальная проблема кроется в том, что при вызове метода из объекта этот объект обычно «принимает участие» в работе метода: метод может использовать поля и другие методы объекта. Когда мы описываем метод в классе, то определяем команды, выполняемые при вызове метода. Но на этапе описания класса объектов еще нет (ведь нельзя создать объект на основе класса, которого не существует — сначала нужно описать класс). Получается, что когда мы описываем метод, то, с одной стороны, команды метода могут быть связаны с объектом, из которого вызывается метод, а с другой стороны, такого объекта на момент описания метода нет. Если речь идет об использовании полей (или других методов) объекта, то действует правило, упомянутое выше: имя поля в коде метода означает поле объекта, из которого вызывается метод. Но далеко не всегда дело ограничивается использованием полей. Часто нужно получить доступ к объекту как таковому. Выход из ситуации находят в использовании специального зарезервированного ключевого слова `this`, обозначающего объект, из которого вызывается метод. Это ключевое слово в теле метода используется так же, как и любая объектная переменная. Просто нужно помнить, что речь идет о том объекте, из которого вызывается метод, содержащий данное ключевое слово.

Допустим, в программе описан такой класс:

```
class MyClass{
   public int code;
   public int get(){
      return code;
   }
}
```

В этом классе описано целочисленное поле `code` и метод `get()`, который результатом возвращает значение поля `code`. В описании метода

get() ссылка на поле дается его названием code. Вместе с тем мы могли бы использовать и полную ссылку this.code, в которой объект, из которого вызывается метод, явно указан с помощью ключевого слова this:

```
class MyClass{
   public int code;
   public int get(){
      return this.code;
   }
}
```

В данном случае использование ключевого слова this не является обязательным. Но бывают ситуации, когда без него не обойтись. Рассмотрим еще один способ описания класса:

```
class MyClass{
   public int code;
   public void set(int n){
      code=n;
   }
}
```

Теперь в классе MyClass кроме поля code описан метод set() с целочисленным аргументом n, который присваивается значением полю code. Команда code=n в теле метода пока особых вопросов не вызывает. Но давайте представим, что мы захотели назвать аргумент метода set() точно так же, как название поля code. Сразу возникает вопрос: если мы в теле используем идентификатор code, то что подразумевается — поле или аргумент? Ответ однозначный и состоит в том, что в таком случае идентификатор code будет отождествляться с аргументом метода. А как же тогда обратиться к одноименному полю? И вот здесь пригодится ключевое слово this:

```
class MyClass{
   public int code;
   public void set(int code){
      this.code=code;
   }
}
```

В теле метода `set()` инструкция `this.code` означает поле `code` объекта, из которого вызывается метод, а идентификатор `code` означает аргумент метода.

Небольшой пример, в котором используется ключевое слово `this`, представлен в листинге 6.8.

Листинг 6.8. Ключевое слово this

```
using System;
// Класс:
class MyClass{
   // Закрытое целочисленное поле:
   private int code;
   // Открытый метод:
   public int get(){
      // Использовано ключевое слово this:
      return this.code;
   }
   // Открытый метод:
   public void set(int code){
      // Использовано ключевое слово this:
      this.code=code;
   }
   // Конструктор:
   public MyClass(int code){
      // Использовано ключевое слово this:
      this.code=code;
      // Использовано ключевое слово this:
      Console.WriteLine("Создан объект: "+this.get());
   }
}
// Класс с главным методом:
class UsingThisDemo{
   // Главный метод:
```

```
   static void Main(){
      // Создание объекта:
      MyClass obj=new MyClass(100);
      // Присваивание значения полю:
      obj.set(200);
      // Проверка значения поля:
      Console.WriteLine("Новое значение: "+obj.get());
   }
}
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 6.8)

```
Создан объект: 100
Новое значение: 200
```

Пример очень простой. В классе `MyClass` есть закрытое целочисленное поле `code`. Значение полю присваивается методом `set()`, аргумент которого имеет такое же название, как и имя поля. Поэтому в теле метода при обращении к полю используется полная ссылка `this.code`. То же замечание справедливо и для конструктора, аргумент которого назван как `code`. Поэтому в теле конструктора идентификатор `code` обозначает аргумент, а инструкция `this.code` является обращением к полю объекта, для которого вызван конструктор. Ключевое слово `code` использовано еще в нескольких местах, хотя острой необходимости в этом не было (имеется в виду инструкция `this.code` в теле метода `get()` и инструкция `this.get()` в теле конструктора). В главном методе создается объект `obj` класса `MyClass`. При вызове конструктора значение поля созданного объекта отображается в консольном окне. Затем с помощью метода `set()` полю объекта присваивается новое значение, а с помощью метода `get()` выполняется проверка значения поля.

Ключевое слово `this` используется и в иных ситуациях. Например, при перегрузке конструкторов в классе ключевое слово `this` с круглыми скобками (и аргументами, если необходимо) используется для вызова одной версии конструктора в другой версии конструктора. Выражение вида `this(аргументы)` указывается через двоеточие после закрывающей круглой скобки в описании конструктора. Оно означает выполнение программного кода той версии конструктора, которая соответствует

аргументам, указанным с ключевым словом this. Общий шаблон выглядит так (жирным шрифтом выделены основные элементы шаблона):

```
public Конструктор(аргументы): this(аргументы){
   // Команды в теле конструктора
}
```

Для большей наглядности рассмотрим небольшой пример, представленный в листинге 6.9.

Листинг 6.9. Конструкторы и ключевое слово this

```
using System;
// Класс с перегрузкой конструкторов:
class MyClass{
   // Целочисленные поля:
   public int alpha;
   public int bravo;
   // Конструктор с одним аргументом:
   public MyClass(int a){
      // Сообщение в консольном окне:
      Console.WriteLine("Конструктор с одним аргументом");
      // Значения полей:
      alpha=a;
      bravo=alpha;
      // Отображение значений полей:
      Console.WriteLine("Оба поля равны "+alpha);
   }
   // Конструктор с двумя аргументами:
   public MyClass(int a, int b): this(a){
      // Сообщение в консольном окне:
      Console.WriteLine("Конструктор с двумя аргументами");
      // Значение второго поля:
      bravo=b;
      // Отображение значений полей:
      Console.WriteLine("Поля "+alpha+" и "+bravo);
```

```
   }
   // Конструктор без аргументов:
   public MyClass(): this(400,500){
      // Сообщение в консольном окне:
      Console.WriteLine("Конструктор без аргументов");
      // Отображение значений полей:
      Console.WriteLine("Значения "+alpha+" и "+bravo);
   }
}
// Класс с главным методом:
class ConstrAndThisDemo{
   // Главный метод:
   static void Main(){
      // Вызов конструктора с одним аргументом:
      MyClass A=new MyClass(100);
      Console.WriteLine();
      // Вызов конструктора с двумя аргументами:
      MyClass B=new MyClass(200,300);
      Console.WriteLine();
      // Вызов конструктора без аргументов:
      MyClass C=new MyClass();
   }
}
```

Ниже показано, как выглядит результат выполнения программы.

Результат выполнения программы (из листинга 6.9)

```
Конструктор с одним аргументом
Оба поля равны 100

Конструктор с одним аргументом
Оба поля равны 200
```

```
Конструктор с двумя аргументами
Поля 200 и 300


Конструктор с одним аргументом
Оба поля равны 400
Конструктор с двумя аргументами
Поля 400 и 500
Конструктор без аргументов
Значения 400 и 500
```

В примере описан класс `MyClass` с двумя целочисленными полями `alpha` и `bravo`. Кроме них в классе есть три версии конструктора: с одним аргументом, с двумя аргументами и без аргументов. Конструктор с одним аргументом простой: появляется сообщение о том, что у конструктора один аргумент, обоим полям присваиваются одинаковые значения (определяется аргументом конструктора), и присвоенное значение отображается в консольном окне. Команда `MyClass A=new MyClass(100)` в главном методе программы дает пример использования этого конструктора.

Конструктор с двумя аргументами описан так, что сначала вызывается версия конструктора с одним аргументом (первый из двух аргументов, переданных конструктору с двумя аргументами), а затем появляется сообщение о том, что вызван конструктор с двумя аргументами, второму полю присваивается новое значение, и конечные значения полей отображаются в консольном окне. Пример использования конструктора с двумя аргументами дается командой `MyClass B=new MyClass(200,300)` в главном методе программы.

Наконец, конструктор без аргументов описан так, что сначала выполняется код конструктора с двумя аргументами `300` и `400`. Выполнение этого конструктора, как мы уже знаем, начинается с выполнения кода конструктора с одним аргументом. И только в самом конце выполняются команды, непосредственно описанные в теле конструктора без аргументов. Конструктор без аргументов используется при создании объекта командой `MyClass C=new MyClass()` в главном методе программы.

Резюме

Что ей надо, я тебе нотом скажу.

из к/ф «Бриллиантовая рука»

- Класс представляет собой шаблон, на основе которого создаются объекты. Описание класса начинается с ключевого слова `class`, после которого указывают имя класса, а в блоке из фигурных скобок описывают поля и методы класса.
- Поля описываются так же, как и объявляются переменные: указывается идентификатор типа и имя поля. При описании методов указываются идентификатор типа возвращаемого результата, имя метода, список аргументов и команды, формирующие тело метода (выделяются фигурными скобками).
- Поля и методы класса называются членами класса. По умолчанию члены класса являются закрытыми — они доступны только внутри кода класса. Чтобы сделать поле или метод открытым (доступным вне пределов класса), его описывают со спецификатором уровня доступа `public`. Закрытые члены класса можно описывать с идентификатором уровня доступа `private`.
- Методы могут перегружаться: в классе может быть описано несколько версий метода с одним и тем же именем, но разными аргументами. Решение о том, какая версия метода используется, принимается на основе команды вызова метода с учетом количества и типа аргументов, фактически переданных методу.
- Конструктор является методом, автоматически вызываемым при создании объекта. Конструктор описывается особым образом: имя конструктора совпадает с именем класса, он не возвращает результат, и идентификатор типа результата для конструктора не указывается. Обычно конструктор описывается с ключевым словом `public`. У конструктора могут быть аргументы, и конструктор можно перегружать (в классе может быть несколько версий конструктора).
- Если в классе конструктор не описан, то используется конструктор по умолчанию: у него нет аргументов, и он не выполняет никаких дополнительных действий. Если в классе описана хотя бы одна версия конструктора, то конструктор по умолчанию больше не доступен.

- Деструктор является методом, автоматически вызываемым при удалении объекта из памяти. Имя деструктора получается объединением символа тильды ~ и названия класса. У деструктора нет аргументов, он не возвращает результат, и идентификатор типа результата для деструктора не указывается. В классе может быть только один деструктор.
- Для создания объекта класса используют оператор `new`, после которого указываются имя класса и аргументы (в круглых скобках), которые передаются конструктору. Ссылка на созданный объект записывается в объектную переменную. Объектная переменная объявляется так же, как и переменная простого типа, только в качестве идентификатора типа указывается имя класса.
- Обращение к нестатическим полям и методам выполняется с указанием объекта: после имени объекта через точку следует название поля или имя метода (с аргументами в круглых скобках или пустыми круглыми скобками, если аргументы не передаются).
- Статические поля и методы описываются с ключевым словом `static`. Статические поля и методы «не привязаны» к объекту и существуют, даже если ни один объект класса не создан. Обращение к статическим полям и методам выполняется так: имя класса, точка и название статического поля или метода (с аргументами или без). Если поле или метод используются в классе, в котором они описаны, то имя класса можно не указывать.
- Ключевое слово `this` в теле методов (в том числе конструкторов и деструкторов) может использоваться для обозначения объекта, из которого вызывается метод. Это же ключевое слово используется как команда вызова в одной версии конструктора другой версии конструктора. В таком случае в описании конструктора после закрывающей круглой скобки указываются двоеточие, ключевое слово `this` и круглые скобки, в которых можно указать аргументы, которые передаются вызываемой версии конструктора. Вызываемая версия конструктора определяется по переданным ей аргументам.

Задания для самостоятельной работы

Перезагрузка завершена. Счастливого пути!

из к/ф «Гостья из будущего»

1. Напишите программу с классом, в котором есть закрытое символьное поле и три открытых метода. Один из методов позволяет присвоить значение полю. Еще один метод при вызове возвращает результатом код символа. Третий метод позволяет вывести в консольное окно символ (значение поля) и его код.

2. Напишите программу с классом, у которого есть два символьных поля и метод. Он не возвращает результат, и у него нет аргументов. При вызове метод выводит в консольное окно все символы из кодовой таблицы, которые находятся «между» символами, являющимися значениями полей объекта (из которого вызывается метод). Например, если полям объекта присвоены значения `'A'` и `'D'`, то при вызове метода в консольное окно должны выводиться все символы от `'A'` до `'D'` включительно.

3. Напишите программу с классом, у которого есть два целочисленных поля. В классе должны быть описаны конструкторы, позволяющие создавать объекты без передачи аргументов, с передачей одного аргумента и с передачей двух аргументов.

4. Напишите программу с классом, у которого есть символьное и целочисленное поле. В классе должны быть описаны версии конструктора с двумя аргументами (целое число и символ — определяют значения полей), а также с одним аргументом типа `double`. В последнем случае действительная часть аргумента определяет код символа (значение символьного поля), а дробная часть (с учетом десятых и сотых) определяет значение целочисленного поля. Например, если аргументом передается число `65.1267`, то значением символьного поля будет символ `'A'` с кодом `65`, а целочисленное поле получит значение `12` (в дробной части учитываются только десятые и сотые).

5. Напишите программу с классом, у которого есть закрытое целочисленное поле. Значение полю присваивается с помощью открытого метода. Методу аргументом может передаваться целое число, а также метод может вызываться без аргументов. Если метод вызывается без аргументов, то поле получает нулевое значение. Если метод вызывается с целочисленным аргументом, то это значение присваивается полю. Однако

если переданное аргументом методу значение превышает `100`, то значением полю присваивается число `100`. Предусмотрите в классе конструктор, который работает по тому же принципу, что и метод для присваивания значения полю. Также в классе должен быть метод, позволяющий проверить значение поля.

6. Напишите программу с классом, в котором есть два закрытых целочисленных поля (назовем их `max` и `min`). Значение поля `max` не может быть меньше значения поля `min`. Значения полям присваиваются с помощью открытого метода. Метод может вызываться с одним или двумя целочисленными аргументами. При вызове метода значения полям присваиваются так: сравниваются текущие значения полей и значения аргумента или аргументов, переданных методу. Самое большое из значений присваивается полю `max`, а самое маленькое из значений присваивается полю `min`. Предусмотрите конструктор, который работает по тому же принципу, что и метод для присваивания значений полям. В классе также должен быть метод, отображающий в консольном окне значения полей объекта.

7. Напишите программу с классом, в котором есть два поля: символьное и текстовое. В классе должен быть перегруженный метод для присваивания значений полям. Если метод вызывается с символьным аргументом, то соответствующее значение присваивается символьному полю. Если метод вызывается с текстовым аргументом, то он определяет значение текстового поля. Методу аргументом также может передаваться символьный массив. Если массив состоит из одного элемента, то он определяет значение символьного поля. В противном случае (если в массиве больше одного элемента) из символов массива формируется текстовая строка и присваивается значением текстовому полю.

8. Напишите программу с классом, в котором есть закрытое статическое целочисленное поле с начальным нулевым значением. В классе должен быть описан статический метод, при вызове которого отображается текущее значение статического поля, после чего значение поля увеличивается на единицу.

9. Напишите программу с классом, в котором есть статические методы, которым можно передавать произвольное количество целочисленных аргументов (или целочисленный массив). Методы, на основании переданных аргументов или массива, позволяют вычислить: наибольшее значение, наименьшее значение, а также среднее значение из набора чисел.

10. Напишите программу со статическим методом для вычисления косинуса. Используйте формулу $\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots + \frac{(-1)^n x^{2n}}{(2n)!}$. В классе также должны быть статические методы для вычисления гиперболического синуса $\mathrm{sh}(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots + \frac{x^{2n+1}}{(2n+1)!}$ и гиперболического косинуса $\mathrm{ch}(x) = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \ldots + \frac{x^{2n}}{(2n)!}$.

Глава 7
РАБОТА С ТЕКСТОМ

> Так, значит, русский язык знаем. Зачем потребовалось скрывать?
>
> *из к/ф «Кин-дза-дза»*

С текстом мы уже имели дело много раз. Вместе с тем наше знакомство было скорее поверхностным. В этой главе мы рассмотрим некоторые технические вопросы, связанные в первую очередь со способом реализации текстовых значений в языке C#. Также мы узнаем о некоторых методах, приемах и подходах, полезных (или просто незаменимых) при работе с текстовыми значениями. В частности, далее речь пойдет о следующем:

- познакомимся с классом `String`, объекты которого используются для реализации текстовых значений;
- проанализируем особенности создания текстовых объектов;
- исследуем основные операции, выполняемые с текстом;
- познакомимся с методами, предназначенными для работы с текстом;
- узнаем о методе `ToString()`, который играет особую роль при преобразовании объектов в текст.

Теперь настало время поближе познакомиться с классом, объекты которого используются для реализации текстовых значений.

Класс String

— Вот вы, папример, кто?
— Я Марк Туллий Цицерон!
— Я не в этом смысле! По пачпорту кто?
— Анна Петровпа Спешнева по паспорту.

из к/ф «О бедном гусаре замолвите слово»

Ранее в программах мы объявляли переменные «типа» `string` и значениями таким переменным присваивали текст. В действительности текстовые значения реализуются как объекты библиотечного класса `String` из пространства имен `System`. Идентификатор `string`, используемый нами для обозначения «текстового типа», в действительности является псевдонимом для инструкции `System.String` — то есть фактически это название класса `String` с учетом пространства имен, к которому относится класс.

Как и для всех прочих объектов в языке C#, для работы с объектами класса `String` используются объектные переменные. Каждый раз, когда мы объявляли переменную типа `string`, на самом деле мы объявляли объектную переменную класса `String`. Объекты класса `String` при этом явно не создавались, хотя они и присутствовали незримо «за кулисами». Например, мы переменным типа `string` значениями присваивали текстовые литералы (текст, заключенный в двойные кавычки). Внешне это выглядело так, как если бы переменной некоторого «текстового» типа присваивалось текстовое значение. В действительности текстовые литералы реализуются как объекты класса `String`. Поэтому, когда мы присваиваем текстовый литерал переменной типа `string`, то объектной переменной класса `String` присваивается ссылка на объект класса `String`, через который реализован текстовый литерал. Не очень тривиальным образом выполняются и прочие операции с текстовыми значениями. Мы их проанализируем и посмотрим на ситуацию под несколько иным углом зрения. Но это будет немного позже. Сейчас же мы остановимся на «технических» подробностях, связанных с реализацией объекта класса `String`.

Как отмечалось выше, при работе с текстом приходится иметь дело с объектной переменной класса `String`, которая ссылается на объект с текстом. Схематически эта ситуация проиллюстрирована на рис. 7.1.

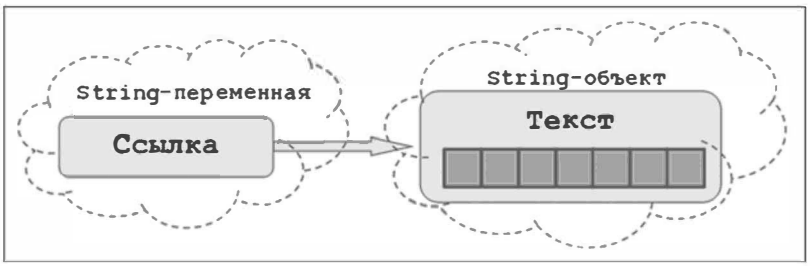


Рис. 7.1. Схема реализации текста

Реальный текст «спрятан» в объекте (откуда он взялся — это вопрос другой).

ПОДРОБНОСТИ

В действительности внутри текстового объекта содержится символьный массив. Элементы этого массива — символы, которые формируют текст. В принципе, до этого массива можно «добраться» (с помощью указателей).

Особенность текстового объекта в том, что после создания его нельзя изменить (здесь речь идет об объекте, а не об объектной переменной). На первый взгляд, данное утверждение может показаться странным — ведь раньше мы с успехом изменяли текстовые значения, реализованные с помощью переменных типа `string`. Но противоречия здесь нет. Мы изменяли значение объектной переменной, а не самого объекта. Например, предположим, что в программе объявлена текстовая переменная `text`:

```
string text="текст";
```

Что происходит в этом случае? Объявляется объектная переменная `text`, и значением ей присваивается ссылка на объект класса `String`, в который записано текстовое значение `"текст"`. Текстовая переменная `text` *не содержит* значение `"текст"` — она на него *ссылается*!

Теперь представим, что выполняется следующая команда:

```
text+="еще один текст";
```

Такого типа команды мы всегда интерпретировали как добавление к текущему текстовому значению переменной `text` текста `"еще один текст"`, указанного справа в выражении. Конечный результат таким и будет: если после выполнения команды мы проверим значение переменной `text`, то это будет объединение текстовых строк `"текст"`

и "еще один текст". Но достигается данный результат не так просто, как может показаться на первый взгляд.

Мы уже знаем, что сначала переменная text ссылается на текстовый объект с текстом "текст". Команда text+="еще один текст" выполняется следующим образом. Текст "еще один текст" не добавляется в объект, на который ссылается переменная text. Вместо этого создается новый текстовый объект. В него записывается текст, который получается объединением двух текстовых значений: текста из объекта, на который ссылается переменная text, и текста "еще один текст" из правой части команды присваивания. Ссылка на этот новый объект записывается в переменную text.

НА ЗАМЕТКУ

То есть текст "еще один текст" не дописывается в текущий объект, на который ссылается переменная text, а создается новый текстовый объект с новым значением, и ссылка на этот новый объект записывается в переменную text. В результате создается иллюзия, что к текущему текстовому значению было дописано значение "еще один текст".

ПОДРОБНОСТИ

Если быть до конца откровенным, то нужно отметить, что способ изменить текстовый объект все же существует. Для этого придется использовать указатели. С их помощью удается «проникнуть» внутрь текстового объекта и добраться до массива, в котором хранятся символы, формирующие текстовое значение. И эти символы можно поменять. Все это мы обсудим, когда будем знакомиться с указателями.

Создание текстового объекта

Болтают все. Не на всех пишут.

из к/ф «О бедном гусаре замолвите слово»

Теперь остановимся на способах создания объектов класса String. Один путь нам уже известен — можно воспользоваться текстовым литералом. Но существуют и иные возможности.

ПОДРОБНОСТИ

Еще раз напомним, что текстовый литерал (то есть текст в двойных кавычках) реализуется как объект класса `String`. Значение текстового литерала, которое мы воспринимаем именно как текст, в действительности является ссылкой на объект, в котором этот текст «спрятан». Поэтому когда объектной переменной типа `string` (или `String`) присваивается значение, то на самом деле этой переменной присваивается ссылка на текстовый объект.

Способы создания текстовых объектов определяются версиями конструктора, которые есть у класса `String`. Фактически речь идет о том, какие аргументы можно передавать конструктору класса `String`. Все варианты рассматривать не будем, но основные перечислим. Итак, текстовый объект можно создать на основе символьного массива. В этом случае из элементов символьного массива формируется текст, и этот текст записывается в созданный текстовый объект. Пример создания текста на основе символьного массива представлен ниже:

```
char[] symbs={'Я','з','ы','к',' ','C','#'};
String txt=new String(symbs);
```

В данном случае текстовый объект создается на основе символьного массива `symbs`. Для создания объекта вызывается конструктор класса `String`, и аргументом ему передается символьный массив. Поскольку массив `symbs` создавался с использованием списка инициализации `{'Я','з','ы','к',' ','C','#'}`, то в объект, на который ссылается переменная `txt`, будет записан текст `"Язык C#"`.

Если при создании текста на основе символьного массива конструктору класса `String`, кроме имени массива, передать еще и два целочисленных аргумента, то они будут определять индекс начального элемента в массиве для формирования текста и количество символов, используемых в тексте. Например, используется такая последовательность команд:

```
char[] symbs={'М','ы',' ','и','з','у','ч','а','е','м',' ','C','#'};
String txt=new String(symbs,3,9);
```

Исходный массив, на основе которого создается текстовый объект, инициализируется списком `{'М','ы',' ','и','з','у','ч','а','е','м',' ','C','#'}`. При создании текстового объекта конструктору класса `String` первым аргументом передается имя массива `symbs`. Есть еще второй и третий аргументы. Второй аргумент `3`

означает, что из массива `symbs` при формировании текста нужно брать символы, начиная с элемента с индексом `3` (четвертый по порядку элемент в массиве со значением `'и'`). Третий аргумент `9` означает, что всего нужно взять `9` символов. В итоге получается, что созданный текстовый объект содержит текст `"изучаем C"`.

Если конструктору класса `String` передать символьное значение и целое число, то в созданном текстовом объекте текст будет сформирован повторением символа (первый аргумент) в количестве, определяемом числом (второй аргумент). Пример такой ситуации приведен ниже:

```
String txt=new String('*',5);
```

В результате создается текстовый объект с текстом `"*****"` (символ `'*'` повторяется `5` раз). Иногда такой подход бывает удобен.

НА ЗАМЕТКУ

Существуют и другие способы создания текстовых объектов. В частности, аргументом конструктору можно передавать указатель на символьный или целочисленный (байтовый) массив с кодами символов, с дополнительными целочисленными аргументами или без них. Указателям посвящена отдельная глава во второй части книги.

Копию текстового объекта можно создать с помощью статического метода `Copy()` класса `String`. Копируемая строка передается аргументом методу `Copy()`. Результатом метода возвращается копия этой строки.

Небольшой пример, в котором текстовые объекты создаются разными способами, представлен в листинге 7.1.

Листинг 7.1. Создание текстовых объектов

```
using System;
class StringDemo{
   static void Main(){
      // Символьный массив:
      char[] symbs={'Я','з','ы','к',' ','C','#'};
      Console.WriteLine("Символьный массив:");
      // Отображение содержимого символьного массива:
      Console.WriteLine(symbs);
      // Текстовый объект создается
```

```
// на основе символьного массива:
String A=new String(symbs);
// Проверка значения текстового объекта:
Console.WriteLine("A: \"{0}\"", A);
// Текстовый объект создается на основе
// нескольких элементов из символьного массива:
String B=new String(symbs,1,5);
// Проверка значения текстового объекта:
Console.WriteLine("B: \"{0}\"", B);
// Текстовый объект создается как
// последовательность одинаковых символов:
String C=new String('ы',7);
// Проверка значения текстового объекта:
Console.WriteLine("C: \"{0}\"", C);
// Создание текстового объекта путем вызова
// статического метода с двумя символьными
// аргументами:
A=getText('A','H');
// Проверка значения текстового объекта:
Console.WriteLine("A: \"{0}\"", A);
// Создание текстового объекта путем вызова
// статического метода с двумя символьными
// аргументами:
B=getText('H','A');
// Проверка значения текстового объекта:
Console.WriteLine("B: \"{0}\"", B);
// Создание текстового объекта путем вызова
// статического метода с аргументом – символьным
// массивом:
C=getText(symbs);
// Проверка значения текстового объекта:
Console.WriteLine("C: \"{0}\"", C);
```

```
    }
    // Статический метод для создания текстового объекта
    // на основе символьного массива:
    static String getText(char[] symbs){
        // Локальная текстовая переменная:
        String txt="";
        // Перебор элементов в массиве:
        for(int k=0; k<symbs.Length; k++){
            // Дописывание символа к текстовой строке:
            txt+=symbs[k];
        }
        // Результат метода:
        return txt;
    }
    // Статический метод для создания текстового объекта
    // на основе двух символьных значений:
    static String getText(char a, char b){
        // Локальная текстовая переменная:
        String txt="";
        // Локальная символьная переменная:
        char s=a;
        // Формирование текстовой строки:
        while(s<=b){
            // Дописывание символа к текстовой строке:
            txt+=s;
            // Следующий символ для строки:
            s++;
        }
        // Результат метода:
        return txt;
    }
}
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 7.1)

```
Символьный массив:
Язык C#
A: "Язык C#"
B: "зык C"
C: "ыыыыыы"
A: "ABCDEFGH"
B: ""
C: "Язык C#"
```

В программе мы несколькими способами создаем текстовые объекты и затем проверяем их значение (текст, содержащийся в этих объектах). Также в программе описывается две версии статического метода `getText()`, который результатом возвращает текстовое значение. Одна версия метода подразумевает, что аргументом передается символьный массив, а результатом метод возвращает текстовую строку, сформированную из символов массива. Здесь мы фактически продублировали конструктор класса `String`, позволяющий создавать текстовые объекты на основе символьных массивов. Еще одна версия метода `getText()` подразумевает, что аргументами методу передаются два символьных значения (предполагается, что код первого символа-аргумента не больше кода второго символа-аргумента, хотя это и не принципиально). Как работают эти методы? Идея достаточно простая. В теле метода (в обеих версиях) объявляется локальная текстовая переменная `txt`, начальное значение которой — пустая текстовая строка. Дальше все зависит от того, какой аргумент или аргументы переданы методу. Если методу передан символьный массив, то запускается оператор цикла `for`, в котором последовательно перебираются элементы символьного массива-аргумента и каждый символ дописывается к текстовой строке (команда `txt+=symbs[k]` в теле оператора цикла `for`, где через `symbs` обозначен переданный аргументом методу символьный массив). По завершении оператора цикла значение из переменной `txt` возвращается результатом метода. Если же аргументами методу `getText()` переданы два символьных значения, то алгоритм выполнения метода немного другой. Локальной символьной переменной `s` присваивается значение первого символьного аргумента `a`. После этого запускается оператор цикла

while, в котором проверяется условие s<=b (через b обозначен второй символьный аргумент). Поскольку s и b — символьные переменные, то при вычислении значения выражения s<=b они автоматически приводятся к целочисленному формату. В итоге сравниваются коды символов. Фактически оператор цикла выполняется до тех пор, пока символ из переменной s в кодовой таблице (или алфавите, если речь идет о буквах) находится до символа из аргумента b. За каждый цикл командой txt+=s к текстовой строке дописывается очередной символ, а затем командой s++ значение переменной s «увеличивается на единицу» — получаем следующий символ в кодовой таблице (следующую букву в алфавите). После того как текстовая строка сформирована, она возвращается результатом метода.

НА ЗАМЕТКУ

Если аргументом методу getText() передать два символа, причем код первого символа будет больше кода второго символа (то есть буквы-аргументы указаны не в алфавитном порядке), то результатом метод возвращает пустую текстовую строку. Причина в том, что при первой проверке условия s<=b в условном операторе оно окажется ложным, и команды в теле оператора цикла выполняться не будут. В итоге переменная txt, возвращаемая как результат метода, будет ссылаться на свое начальное значение — на пустую текстовую строку.

В главном методе программы командой char[] symbs={'Я','з','ы','к',' ','С','#'} создается и инициализируется символьный массив, который мы используем для создания текстовых объектов. Но сначала проверяется содержимое этого массива, для чего массив symbs передается аргументом методу WriteLine(). В этом случае содержимое символьного массива печатается в консольном окне как текст.

ПОДРОБНОСТИ

Метод WriteLine() определен так, что если ему аргументом передается имя символьного массива, то в консольное окно выводятся символы из массива. Например, если symbs — символьный массив, то в результате выполнения команды Console.WriteLine(symbs) отображается содержимое массива symbs. Но если мы воспользуемся командой Console.WriteLine(""+symbs) (аргументом методу передается сумма текстового литерала и имени символьного массива) или командой Console.WriteLine("{0}", symbs), то в консольном окне появится не содержимое массива symbs, а сообщение

System.Char[]. Дело в том, что в первом случае (команда Console.WriteLine(symbs)) вызывается версия метода WriteLine() для обработки символьного массива, переданного аргументом. А при выполнении команд Console.WriteLine(""+symbs) и Console.WriteLine("{0}", symbs) выполняется попытка преобразовать значение переменной symbs к текстовому формату. В этом случае вызывается метод ToString() (обсуждается в одном из следующих разделов). Метод возвращает текстовое представление для преобразуемого объекта (в данном случае это symbs). По умолчанию метод возвращает идентификатор с названием класса для объекта. Для символьного массива это идентификатор System.Char[].

Первый текстовый объект создается командой String A=new String(symbs) на основе символьного массива. Содержимое текстового объекта проверяем командой Console.WriteLine("A: \"{0}\"", A).

НА ЗАМЕТКУ

Командой Console.WriteLine("A: \"{0}\"", A) отображается текстовая строка "A: \"{0}\"", в которой вместо блока {0} подставляется значение текстовой переменной A (точнее, подставляется текст из объекта, на который эта переменная ссылается). Причем отображаемый текст заключается в двойные кавычки. Двойные кавычки в текстовую строку добавляем с помощью инструкции \".
Аналогичным способом проверяются значения прочих текстовых объектов.

В данном случае текст получается объединением в одну строку символов из массива. Еще один объект создается командой String B=new String(symbs,1,5). Теперь текстовый объект создается на основе того же массива symbs, но используются не все символы массива, а только символы, начиная со второго (с индексом 1 — второй аргумент), и используется 5 символов (третий аргумент).

Текстовый объект, который создается командой String C=new String('ы',7), содержит текст из семи букв 'ы'.

В результате выполнения команды A=getText('A','H') создается текстовый объект, который состоит из символов от 'A' до 'H' включительно, а ссылка на этот объект записывается в переменную A.

При выполнении команды B=getText('H','A') в переменную B записывается ссылка на текстовый объект с пустой текстовой строкой,

поскольку в данном случае аргументы метода `getText()` указаны не в алфавитном порядке.

Наконец, с помощью команды `C=getText(symbs)` на основе массива `symbs` создается текстовый объект, а ссылка на него присваивается значением переменной `C`.

Операции с текстовыми объектами

> Ну, вы мой полк не марайте. Мои орлы газет не читают, книг в глаза пе видели — никаких идей пе имеют!
>
> *из к/ф «О бедном гусаре замолвите слово»*

При выполнении базовых операций с текстовыми значениями полезно помнить, что на базовом уровне текст реализуется как символьный массив (правда, еще нужно уметь добраться до этого массива). В этом смысле некоторые методы работы с текстовыми объектами неожиданными не станут. Например, узнать количество символов в тексте можно с помощью свойства `Length`. Скажем, если `txt` — объектная переменная класса `String`, то значением выражения `txt.Length` является количество символов в тексте, содержащемся в объекте, на который ссылается переменная `txt`. Причем поскольку текстовые литералы реализуются в виде объектов класса `String`, то свойство `Length` есть и у текстовых литералов. Поэтому выражение `"текст".Length` имеет смысл. Значение этого выражения равно `5` — это количество символов в тексте `"текст"`. Если нам нужен отдельный символ из текста, то мы можем проиндексировать текст: после текстовой переменной или текстового литерала в квадратных скобках указывается индекс символа. Как и в случае с одномерным массивом, индексация начинается с нуля. То есть индекс первого символа в тексте равен `0`, а индекс последнего символа в тексте на единицу меньше количества символов в тексте. Так, если `txt` является текстовой переменной, то значением выражения `txt[0]` является первый символ в тексте, значением выражения `txt[1]` является второй символ в тексте, а значением выражения `txt[txt.Length-1]` является последний символ в тексте.

> **НА ЗАМЕТКУ**
>
> Нужно учесть, что `txt.Length` — это количество символов в тексте (на который ссылается переменная `txt`), а индекс последнего символа на единицу меньше количества символов в тексте.

Индексировать можно и текстовые литералы. Так, значением выражения `"текст"[2]` является символ `'к'` — третья по счету (с индексом 2) буква в тексте `"текст"`. Пример операций, связанных с индексированием текстовых значений и перебором символов в тексте, представлен в программе в листинге 7.2.

Листинг 7.2. Перебор символов в тексте

```
using System;
class IndexingStringDemo{
   static void Main(){
      // Отображение текстового литерала с помощью
      // оператора цикла по коллекции:
      foreach(char s in "Текст"){
         Console.Write(s+" ");
      }
      // Переход к новой строке:
      Console.WriteLine();
      // Отображение текстового литерала с помощью
      // оператора цикла:
      for(int k=0; k<"Текст".Length; k++){
         Console.Write("Текст"[k]+"_");
      }
      // Переход к новой строке:
      Console.WriteLine();
      // Текстовая переменная:
      String A="Изучаем C#";
      // Отображение текстового значения:
      for(int k=0; k<A.Length; k++){
         Console.Write(A[k]);
      }
      // Переход к новой строке:
      Console.WriteLine();
      // Отображение текста в обратном порядке:
```

```
      showReversed(A);
      // Проверка значения текстового объекта:
      Console.WriteLine(A);
      // Новый текстовый объект (текст в обратном порядке):
      String B=getReversed(A);
      // Отображение текстового значения:
      Console.WriteLine(B);
   }
   // Статический метод для создания текстового объекта,
   // в котором текст записан в обратном порядке:
   static String getReversed(String txt){
      // Локальная текстовая переменная:
      String str="";
      // Формирование текстовой строки:
      foreach(char s in txt){
         // Добавление символа в начало строки:
         str=s+str;
      }
      // Результат метода:
      return str;
   }
   // Статический метод для отображения текста
   // в обратном порядке:
   static void showReversed(String txt){
      // Перебор символов в тексте (в обратном порядке):
      for(int k=txt.Length-1; k>=0; k--){
         // Отображение в консоли символа:
         Console.Write(txt[k]);
      }
      // Переход к новой строке:
      Console.WriteLine();
   }
}
```

Результат выполнения программы будет таким.

Результат выполнения программы (из листинга 7.2)

```
Т е к с т
Т_е_к_с_т_
Изучаем C#
#C меачузИ
Изучаем C#
#C меачузИ
```

В главном методе программы с помощью оператора цикла по коллекции `foreach` посимвольно отображается текстовый литерал `"Текст"`. Переменная `s`, объявленная в операторе цикла, относится к типу `char` (символ из текста). При отображении символов из литерала после каждого символа добавляется пробел.

Еще один пример посимвольного отображения литерала основан на использовании оператора цикла `for`. Символы в тексте (речь идет о текстовом литерале `"Текст"`) в этом случае перебираются с использованием индекса. Индексная переменная `k` пробегает значения от `0` и до верхней границы, которая на единицу меньше количества символов в текстовом литерале. Длину текстового литерала (количество символов) вычисляем с помощью выражения `"Текст".Length`. Отдельный символ в литерале вычисляется инструкцией `"Текст"[k]`. В данном случае после текстового литерала мы просто указали в квадратных скобках индекс символа. При отображении символов в консоли после каждого символа добавляется подчеркивание.

Командой `String A="Изучаем C#"` объявляется и инициализируется текстовая переменная `A`. Мы можем отобразить текст из соответствующего объекта с помощью команды `Console.WriteLine(A)` (то есть просто передав переменную `A` аргументом методу `WriteLine()`). А можем отобразить текст посимвольно, для чего используется оператор цикла `for`. Индексная переменная `k` последовательно перебирает индексы символов в тексте. Начальное значение индекса равно `0`, а количество символов в тексте вычисляется командой `A.Length`. Обращение к отдельному символу выполняется в формате `A[k]`.

В программе описано два статических метода. Метод `showReversed()` при вызове отображает в консоли в обратном порядке текстовую строку,

переданную аргументом методу. Поэтому при выполнении команды `showReversed(A)` текст из переменной `A` (текст из объекта, на который ссылается переменная) отображается в обратном порядке, но сам текстовый объект при этом не меняется.

НА ЗАМЕТКУ

Индексированием текстовой переменной мы можем прочитать символ в строке, но не можем изменить его значение. Проще говоря, если `A` — текстовая переменная и `k` — индекс символа, то мы можем узнать, что это за символ, с помощью инструкции `A[k]`. В данном случае с символом мы обращаемся как с элементом массива. Но присвоить значение «элементу» `A[k]` мы не можем.

ПОДРОБНОСТИ

Аргументом методу `showReversed()` передается текстовая строка (аргумент обозначен как `txt`). В теле массива с помощью оператора цикла перебираются символы из текстового аргумента, но только символы перебираются в обратном порядке. Индексная переменная `k` принимает начальное значение `txt.Length-1`. Это индекс последнего символа в тексте. Оператор цикла выполняется, пока истинно условие `k>=0` (индекс неотрицательный). За каждый цикл значение индексной переменной уменьшается на единицу (команда `k--`). Символы отображаются по одному, все в одной строке (команда `Console.Write(txt[k])` в теле оператора цикла). После завершения отображения текста выполняется переход к новой строке.

При вызове метода `getReversed()` создается новый текстовый объект. Текст в этом объекте записан в обратном порядке по сравнению с объектом, который передавался аргументом методу при вызове. Следовательно, когда выполняется команда `String B=getReversed(A)`, то в текстовую переменную `B` записывается ссылка на вновь созданный объект. В нем текст записан в обратном порядке, если сравнивать с текстом из объекта, на который ссылается переменная `A`.

ПОДРОБНОСТИ

Статическому методу `getReversed()` аргументом передается текст, и результатом метода является также текстовое значение. В теле метода объявляется локальная текстовая переменная `str` с пустой текстовой строкой в качестве начального значения. Затем запускается

оператор цикла по коллекции `foreach`. Символьная (тип `char`) переменная `s` в операторе цикла последовательно принимает значения символов из текстового аргумента `txt`. За каждый цикл командой `str=s+str` считанный из аргумента символ записывается в начало текстовой строки `str`. После завершения оператора цикла сформированная текстовая строка возвращается как результат метода.

Мы уже знаем, что к текстовым значениям может применяться такая арифметическая операция, как сложение. Если складываются (с помощью оператора +) два текстовых значения, то создается новый текстовый объект, который содержит текст, получающийся объединением складываемых текстовых значений. Результатом выражения возвращается ссылка на созданный объект.

НА ЗАМЕТКУ

Объединение текстовых строк называется конкатенацией строк.

Если из двух складываемых операндов лишь один является текстовым, то второй автоматически приводится к текстовому формату (если это возможно), и уже после этого текстовые строки объединяются. Но здесь нужно быть очень аккуратными. Как пример рассмотрим следующий фрагмент кода:

```
String txt="Четыре "+2+2;
```

Вопрос в том, какой текст в итоге присваивается переменной `txt`. Ответ такой: переменной `txt` присваивается текст `"Четыре 22"`. Выражение `"Четыре "+2+2` вычисляется слева направо. Сначала вычисляется значение выражения `"Четыре "+2`. В этом выражении один операнд текстовый, а другой целочисленный. Целочисленный операнд приводится к текстовому типу, складываемые текстовые строки объединяются, и в итоге получается строка `"Четыре 2"`. К этому текстовому значению прибавляется целочисленное значение `2`, то есть вычисляется значение выражения `"Четыре 2"+2`. Второй целочисленный операнд приводится к текстовому типу, и в итоге получаем строку `"Четыре 22"`. Результат будет иным, если при определении значения переменной `txt` воспользоваться следующей командой:

```
String txt="Четыре "+(2+2);
```

В этом случае круглые скобки изменяют порядок выполнения операций: сначала вычисляется значение выражения `2+2` (получаем число `4`),

а затем полученное значение прибавляется к текстовой строке "Четыре ". В итоге переменной txt присваивается текст "Четыре 4".

НА ЗАМЕТКУ

Для объединения строк можно использовать статический метод Concat() класса String. Аргументами методу передаются текстовые строки (или массив из текстовых строк). Результатом метод возвращает текстовую строку, получающуюся объединением аргументов метода (или элементов текстового массива). При объединении текстовых строк полезным может быть и метод Join(). Это еще один статический метод класса String. Аргументами методу передается текстовая строка, являющаяся разделителем при объединении текстовых строк, и массив с объединяемыми текстовыми строками. Результатом метода возвращается текстовая строка, которая получается объединением текстовых элементов массива (второй аргумент метода), и при объединении строк используется разделитель, определяемый первым аргументом метода.

Еще одна операция, имеющая важное прикладное значение — сравнение текстовых строк на предмет равенства/неравенства. Выполняется сравнение строк с помощью операторов == (проверка на равенство текстовых строк) и != (проверка на неравенство текстовых строк). Если A и B — объектные переменные класса String, то результатом выражения A==B является значение true, если объекты, на которые ссылаются переменные A и B, содержат одинаковый текст. Если эти объекты содержат разный текст, то значение выражения A==B равно false.

Значение выражения A!=B равно true в случае, если переменные A и B ссылаются на объекты, содержащие разный текст. Если эти объекты содержат одинаковый текст, то значение выражения A!=B равно false.

ПОДРОБНОСТИ

Если A и B являются объектными и переменными некоторого класса, то в общем случае при вычислении значения выражения A==B или A!=B сравниваются фактические значения переменных — то есть адреса объектов, на которые ссылаются переменные. Поэтому, например, значение выражения A==B равно true, если переменные A и B ссылаются на один и тот же объект. А если объекты физически разные (пускай при этом и совершенно одинаковые), то результатом выражения A==B будет значение false. Подобная ситуация имеет место и при вычислении выражения вида A!=B. Но это в общем случае.

А для текстовых объектов (то есть когда A и B являются объектными переменными класса String) при вычислении выражений A==B и A!=B сравниваются не адреса объектов, а сами объекты. То есть для объектов класса String сравнение с помощью операторов == и != выполняется по-особому. Отметим, что для сравнения текстовых объектов также может использоваться метод Equals().

Благодаря механизму перегрузки операторов, существующему в C#, для объектов пользовательских классов можно изменить способ сравнения объектов.

Скромная программа, в которой в разных ситуациях используется сравнение текстовых значений, представлена в листинге 7.3.

Листинг 7.3. Сравнение текстовых строк

```
using System;
class CompStringDemo{
   // Статический метод для сравнения текстовых строк:
   static bool StrCmp(String X, String Y){
      // Если строки разной длины:
      if(X.Length!=Y.Length) return false;
      // Если строки одинаковой длины:
      for(int k=0; k<X.Length; k++){
         // Если символы в текстовых строках разные:
         if(X[k]!=Y[k]) return false;
      }
      // Результат метода, если строки одинаковой длины
      // и все символы в текстовых строках совпадают:
      return true;
   }
   // Главный метод:
   static void Main(){
      // Символьный массив:
      char[] smb={'Я','з','ы','к',' ','C','#'};
      // Текстовый объект:
      String A="Язык C#";
```

```
      // Отображение текстовой строки:
      Console.WriteLine("A: \"{0}\"", A);
      // Создание нового текстового объекта:
      String B=new String(smb);
      // Отображение текстовой строки:
      Console.WriteLine("B: \"{0}\"", B);
      // Еще один текстовый объект:
      String C="ЯЗЫК C#";
      // Отображение текстовой строки:
      Console.WriteLine("C: \"{0}\"", C);
      Console.WriteLine("Сравнение строк");
      // Сравнение текстовых строк:
      Console.WriteLine("A==B: {0}", A==B);
      Console.WriteLine("StrCmp(A, B): {0}", StrCmp(A, B));
      Console.WriteLine("A==C: {0}", A==C);
      Console.WriteLine("StrCmp(A, C): {0}", StrCmp(A, C));
      Console.WriteLine("B!=C: {0}", B!=C);
      Console.WriteLine("StrCmp(A,\"C#\"): {0}", StrCmp(A,"C#"));
   }
}
```

При выполнении программы получаем такой результат.

Результат выполнения программы (из листинга 7.3)

```
A: "Язык C#"
B: "Язык C#"
C: "ЯЗЫК C#"
Сравнение строк
A==B: True
StrCmp(A, B): True
A==C: False
StrCmp(A, C): False
B!=C: True
StrCmp(A,"C#"): False
```

В этой простой программе с помощью операторов == и != сравнивается несколько текстовых строк. Также в программе есть описание статического метода StrCmp(), предназначенного для сравнения текстовых строк. Анализ программы начнем с кода метода StrCmp().

НА ЗАМЕТКУ

Строго говоря, нет необходимости описывать специальный метод для сравнения текстовых строк. Сравнивать строки можно с помощью операторов == и !=. В программе соответствующий метод приводится как пример реализации алгоритма, который может использоваться для сравнения текстовых значений. Этот же подход, с минимальными изменениями, может быть использован для сравнения массивов на предмет равенства/неравенства.

Метод StrCmp() описан как статический в том же классе, что и главный метод программы. Поэтому метод StrCmp() можно вызывать в методе Main() без указания названия класса. У метода StrCmp() два текстовых аргумента (объектные переменные X и Y класса String). Через эти переменные методу передаются текстовые значения для сравнения. Если тексты одинаковые, то метод должен возвращать результатом логическое значение true. Если сравниваемые тексты отличаются, то результатом метода должно быть значение false. При выполнении кода метода сначала сравнивается длина текстовых значений. Понятно, что если два текста имеют разную длину, то совпадать они не могут. Это, так сказать, формальная проверка. Выполняется она с помощью условного оператора (в упрощенной форме), в котором проверяется условие X.Length!=Y.Length. Если условие истинно, то инструкцией return false выполнение кода метода завершается, а результатом возвращается значение false. Если же условие ложно, то начинает выполняться оператор цикла, в котором посимвольно проверяется содержимое двух текстовых аргументов метода. Здесь мы учли, что раз уж оператор цикла выполняется, то обе текстовые строки имеют одинаковую длину (иначе выполнение метода завершилось бы при выполнении условного оператора). В теле оператора цикла для каждого фиксированного значения индекса k с помощью условного оператора проверяется условие X[k]!=Y[k]. Истинность этого условия означает, что две буквы, находящиеся на одинаковых позициях, отличаются. В таком случае командой return false завершается выполнение кода метода, а результатом метода возвращается значение false. Команда return true в самом конце тела метода выполняется только в том случае, если при

выполнении оператора цикла оказалось, что все буквы (на одинаковых позициях) в сравниваемых текстовых значениях совпадают. Это означает, что текстовые значения одинаковые и результатом метода возвращается значение `true`.

В главном методе создается несколько текстовых объектов, которые затем сравниваются (с помощью операторов == и !=, а также метода `StrCmp()`). Объект `A` создается присваиванием литерала `"Язык C#"`. Объект `B` создается на основе предварительно объявленного и инициализированного символьного массива, но в итоге получается объект с таким же текстовым значением, как и объект `A`. Объект `C` получает значением текстовый литерал `"ЯЗЫК C#"`. От значений объектов `A` и `B` данный текст отличается лишь регистром букв.

НА ЗАМЕТКУ

Напомним, что регистр букв имеет значение: одна и та же буква в разных регистрах (большая и маленькая) интерпретируется как разный символ.

Также стоит заметить, что текстовые объекты реализуются по следующей схеме: объектная переменная ссылается на объект, содержащий текст. Поэтому присваивание текстового значения переменной следует понимать в том смысле, что переменная будет ссылаться на объект, содержащий присвоенный текст. Для удобства, если это не приводит к недоразумениям, мы иногда не будем заострять внимание на таких деталях.

Для сравнения текстовых значений используются выражения `A==B` (равенство текстовых значений в объектах, на которые ссылаются переменные `A` и `B` — результат равен `true`), `A==C` (проверка на предмет совпадения текстовых значений в объектах, на которые ссылаются переменные `A` и `C` — результат равен `false`), `B!=C` (проверка на предмет неравенства текстовых значений в объектах, на которые ссылаются переменные `B` и `C` — результат равен `true`). Аналогичные проверки выполняются с помощью описанного в программе статического метода `StrCmp()`. Как несложно заметить, результаты проверок вполне ожидаемые.

ПОДРОБНОСТИ

Хотя для сравнения текстовых значений обычно используют операторы == и !=, это не единственный способ сравнения текстовых строк. Для сравнения текстовых строк можно использовать метод

Equals() (вообще, диапазон применения метода Equals() более широк, чем сравнение текстовых значений — этот метод используется для сравнения объектов разных классов, в том числе и объектов класса String). У метода есть несколько версий. Имеется статическая версия метода, которая вызывается из класса String. Аргументами методу в таком случае передаются сравниваемые текстовые строки. Результатом метод возвращает логическое значение true (строки совпадают) или false (строки не совпадают). Нестатическая версия метода вызывается из текстового объекта, а еще один текстовый объект передается аргументом методу. Сравнивается текстовое значение для объекта, из которого вызывается метод, и текстовое значение для объекта, переданного аргументом методу. Результатом метода является логическое значение. Например, если A и B являются объектными переменными класса String, то результатом выражения A.Equals(B) или String.Equals(A, B) является значение true, если переменные A и B ссылаются на текстовые объекты с одинаковым текстом. В противном случае результатом является значение false. Текстовые значения по умолчанию сравниваются с учетом состояния регистра (то есть большая и маленькая буквы считаются разными символами). Если нужно провести сравнение текстовых значений без учета состояния регистра (то есть при сравнении текстов большие и маленькие буквы интерпретировать как один и тот же символ), дополнительным аргументом методу Equals() следует передать выражение StringComparison.OrdinalIgnoreCase (константа OrdinalIgnoreCase перечисления StringComparison). Так, при выполнении команд вида A.Equals(B, StringComparison.OrdinalIgnoreCase) или String.Equals(A, B, StringComparison.OrdinalIgnoreCase) сравнение текстовых значений выполняется без учета состояния регистра. Кстати, если использовать вместо константы OrdinalIgnoreCase константу Ordinal, то при сравнении текстовых строк используется режим по умолчанию, при котором регистр букв имеет значение.

Иногда для сравнения текстовых значений без учета состояния регистра можно текстовые значения предварительно перевести в верхний или нижний регистр. Для перевода символов текста в верхний регистр используется метод ToUpper(), а для перевода символов текста в нижний регистр используют метод ToLower() (методы обсуждаются в следующем разделе). Например, чтобы сравнить значения текстовых переменных A и B на предмет равенства без учета состояния регистра, можно использовать выражения A.ToUpper()==B.ToUpper() или A.ToLower()==B.ToLower(). При этом значения переменных A и B не меняются.

Методы для работы с текстом

— А зачем вы мне это говорите?
— Сигнализирую.

из к/ф «Девчата»

Существует большое количество методов, предназначенных для работы с текстовыми объектами. Эти методы позволяют выполнять самые разные операции со строками. Мы очень кратко обсудим наиболее важные и интересные методы.

НА ЗАМЕТКУ

Напомним, что текстовый объект изменить нельзя: после того как текстовый объект создан, содержащийся в нем текст не может быть изменен. Поэтому здесь и далее все операции, в которых упоминается изменение текста, следует понимать в том смысле, что на основе одного текстового объекта создается другой текстовый объект.

Методы, предназначенные для работы с текстом, условно можно разбить на несколько групп в соответствии с типом решаемой задачи. Так, есть методы для выполнения поиска в текстовой строке. Метод `IndexOf()` позволяет выполнять поиск в текстовой строке символа или подстроки. Метод вызывается из объекта текстовой строки, в котором выполняется поиск. Символ или подстрока для поиска передаются аргументом методу. Результатом метод возвращает целое число — индекс первого вхождения в текстовую строку искомого фрагмента или символа. Если при поиске совпадение не найдено (искомая подстрока или символ отсутствуют в исходной текстовой строке), то метод возвращает значение `-1`. Методу можно передать второй целочисленный аргумент. Если так, то этот аргумент определяет индекс символа в строке, начиная с которого выполняется поиск. Третий необязательный целочисленный аргумент метода `IndexOf()` определяет количество элементов в строке, по которым выполняется поиск.

Метод `LastIndexOf()` похож на метод `IndexOf()`, но только поиск совпадений начинается с конца строки. Если аргументом методу `LastIndexOf()` передать подстроку или символ, то результатом метод возвращает индекс последнего вхождения данной подстроки или символа в текст, из которого вызван метод. В случае, если совпадений нет,

метод возвращает значение –1. Второй необязательный целочисленный аргумент метода определяет индекс элемента, начиная с которого выполняется поиск совпадений. Поиск выполняется в обратном порядке — от конца в начало текста. Третий необязательный целочисленный аргумент определяет количество элементов, по которым выполняется поиск.

Методы `IndexOfAny()` и `LastIndexOfAny()` подобны методам `IndexOf()` и `LastIndexOf()` соответственно, но им первым аргументом передается символьный массив. А поиск выполняется на предмет совпадения хотя бы одного из символов из массива с символом в тексте, из которого вызывается метод. Небольшая иллюстрация к использованию методов `IndexOf()` и `LastIndexOf()`, `IndexOfAny()` и `LastIndexOfAny()` представлена в программном коде в листинге 7.4.

Листинг 7.4. Поиск по тексту

```
using System;
class SearchStringDemo{
   static void Main(){
      // Текст, в котором выполняется поиск:
      String txt="Итак, два плюс два и умножить на два равно шести";
      // Отображение текстового значения:
      Console.WriteLine("Исходный текст:");
      Console.WriteLine("\"{0}\"", txt);
      // Текстовая строка для поиска:
      String str="два";
      // Отображение текстового значения:
      Console.WriteLine("Строка для поиска:");
      Console.WriteLine("\"{0}\"", str);
      // Переменная для записи значения индекса:
      int index;
      // Индекс первого вхождения строки в текст:
      index=txt.IndexOf(str);
      Console.WriteLine("Первое совпадение: {0}", index);
      // Индекс второго вхождения строки в текст:
      index=txt.IndexOf(str, index+1);
```

```
Console.WriteLine("Второе совпадение: {0}", index);
// Индекс последнего вхождения строки в текст:
index=txt.LastIndexOf(str);
Console.WriteLine("Последнее совпадение: {0}", index);
// Индекс для начала поиска и количество символов:
int a=7, b=9;
Console.WriteLine("Поиск с {0}-го индекса по {1} символам:", a, b);
// Индекс первого вхождения строки на интервале:
index=txt.IndexOf(str, a, b);
Console.WriteLine("Совпадение на индексе: {0}", index);
// Изменяем количество символов для поиска:
b+=3;
Console.WriteLine("Поиск с {0}-го индекса по {1} символам:", a, b);
index=txt.IndexOf(str, a, b);
Console.WriteLine("Совпадение на индексе: {0}", index);
// Символ для поиска:
char symb='a';
Console.WriteLine("Теперь ищем букву \'{0}\'", symb);
// Поиск первого совпадения:
index=txt.IndexOf(symb);
Console.WriteLine("Первое совпадение: {0}", index);
// Поиск второго совпадения:
index=txt.IndexOf(symb, index+1);
Console.WriteLine("Второе совпадение: {0}", index);
// Последнее совпадение:
index=txt.LastIndexOf(symb);
Console.WriteLine("Последнее совпадение: {0}", index);
// Предпоследнее совпадение:
index=txt.LastIndexOf(symb, index-1);
Console.WriteLine("Предпоследнее совпадение: {0}", index);
// Поиск на интервале:
index=txt.IndexOf(symb, a, b);
```

```
Console.WriteLine("Поиск на интервале индексов от {0} до {1}", a, a+b-1);
Console.WriteLine("Первое совпадение на интервале: {0}", index);
// Последнее совпадение на интервале:
index=txt.LastIndexOf(symb, b, b+1);
Console.WriteLine("Поиск до индекса {0}", b);
Console.WriteLine("Последнее совпадение на интервале: {0}", index);
// Новый символ для поиска:
symb='ы';
Console.WriteLine("Ищем букву \'{0}\'", symb);
// Поиск первого совпадения:
index=txt.IndexOf(symb);
Console.WriteLine("Первое совпадение: {0}", index);
// Массив с буквами для поиска:
char[] s={'ы','а','д'};
// Отображение символов для поиска:
Console.Write("Ищем букву \'{0}\'", s[0]);
for(int k=1; k<s.Length-1; k++){
   Console.Write(", \'"+s[k]+"\'");
}
Console.WriteLine(" или \'"+s[s.Length-1]+"\':");
// Первое совпадение:
index=txt.IndexOfAny(s);
Console.WriteLine("Первое совпадение: {0}", index);
// Второе совпадение:
index=txt.IndexOfAny(s, index+1);
Console.WriteLine("Второе совпадение: {0}", index);
// Последнее совпадение:
index=txt.LastIndexOfAny(s);
Console.WriteLine("Последнее совпадение: {0}", index);
Console.WriteLine("Поиск на интервале индексов от {0} до {1}", a, a+b-1);
// Первое совпадение на интервале:
index=txt.IndexOfAny(s, a, b);
```

```
      Console.WriteLine("Первое совпадение на интервале: {0}", index);
      // Последнее совпадение на интервале:
      index=txt.LastIndexOfAny(s, a+b-1, b);
      Console.WriteLine("Последнее совпадение на интервале: {0}", index);
   }
}
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 7.4)

```
Исходный текст:
"Итак, два плюс два и умножить на два равно шести"
Строка для поиска:
"два"
Первое совпадение: 6
Второе совпадение: 15
Последнее совпадение: 33
Поиск с 7-го индекса по 9 символам:
Совпадение на индексе: -1
Поиск с 7-го индекса по 12 символам:
Совпадение на индексе: 15
Теперь ищем букву 'а'
Первое совпадение: 2
Второе совпадение: 8
Последнее совпадение: 38
Предпоследнее совпадение: 35
Поиск на интервале индексов от 7 до 18
Первое совпадение на интервале: 8
Поиск до индекса 12
Последнее совпадение на интервале: 8
Ищем букву 'ы'
Первое совпадение: -1
Ищем букву 'ы', 'а' или 'д':
```

```
Первое совпадение: 2
Второе совпадение: 6
Последнее совпадение: 38
Поиск на интервале индексов от 7 до 18
Первое совпадение на интервале: 8
Последнее совпадение на интервале: 17
```

В программе текст, в котором выполняется поиск подстрок и символов, записывается в переменную `txt` (значением является литерал `"Итак, два плюс два и умножить на два равно шести"`). В переменную `str` записывается текст `"два"`. Это строка, которую мы будем искать в тексте. Для записи значения индекса, определяющего позицию строки или символа в тексте, объявляется целочисленная переменная `index`.

Первое значение переменной `index` присваивается командой `index=txt.IndexOf(str)`. Это индекс первого вхождения строки `str` в текст из переменной `txt` (значение `6`). После выполнения команды `index=txt.IndexOf(str, index+1)` переменная `index` получает значение `15`. Значение выражения `txt.IndexOf(str, index+1)` — это индекс первого вхождения строки `str` в текст `txt`, начиная с индекса `index+1` включительно. На предыдущем этапе в переменную `index` был записан индекс первого вхождения строки `str` в текст `txt`, если искать с самого начала текста. Значит, следующее совпадение — второе по счету.

Командой `index=txt.LastIndexOf(str)` в переменную вычисляется индекс последнего вхождения строки `str` в текст `txt` (значение `33`).

Далее мы используем две целочисленные переменные (переменная `a` со значением `7` и переменная `b` со значение `9`). Переменные используются в команде `index=txt.IndexOf(str, a, b)`. Значение переменной `index` после этой команды — индекс первого включения строки `str` в текст `txt`, если поиск начинать с индекса со значением `a` (индекс `7`), причем поиск ограничен количеством символов, определяемых значением переменной `b` (просматривается `9` символов). Получается, что поиск идет по тексту `"ва плюс д"`, в котором нет строки `"два"`. Поэтому метод `IndexOf()` возвращает значение `-1`. А вот если мы проделаем ту же операцию, предварительно увеличив значение переменной `b` на `3` (новое значение этой переменной равно `12`), то поиск строки `"два"` будет выполняться в тексте `"ва плюс два "`. Такая строка в тексте есть.

Индекс, определяющий позицию строки, определяется на основе всего текста из переменной txt. Получается значение 15.

В символьную переменную symb записывается символ 'a', который мы будем искать в тексте txt. Первое появление буквы в тексте определяется индексом, который вычисляется командой index=txt.IndexOf(symb). В тексте из переменной txt буква 'a' первый раз встречается на третьей позиции (индекс на единицу меньше и равен 2). Индекс позиции, где буква 'a' появляется второй раз, вычисляется командой index=txt.IndexOf(symb, index+1) (переменная index получает значение 8).

Индекс позиции, в которой имеет место последнее вхождение символьного значения переменной symb в текст txt, вычисляем командой index=txt.LastIndexOf(symb) и получаем значение 38. Индекс предпоследней позиции буквы 'a'(значение переменной symb) вычисляется командой index=txt.LastIndexOf(symb, index-1). Здесь мы учли, что текущее (до выполнения команды) значение переменной index — это индекс позиции, в которой буква 'a' появляется последний раз. Следовательно, индекс предыдущей буквы в тексте txt равен index-1. Метод LastIndexOf() выполняет поиск в направлении начала текста. Поэтому искать индекс предпоследней позиции буквы 'a' в тексте нужно, начиная с позиции с индексом index-1. Предпоследняя позиция буквы 'a' в тексте из переменной txt определяется индексом со значением 35.

Командой index=txt.IndexOf(symb, a, b) вычисляется индекс позиции буквы 'a'(значение переменной symb) в тексте txt, начиная с позиции с индексом 7 (значение переменной a), и проверяется 12 символов (значение переменной b).

ПОДРОБНОСТИ

Если поиск начинается с символа в позиции с индексом a, а количество проверяемых символов равно b, то это означает, что поиск выполняется на интервале индексов от a до a+b-1 включительно. Для значений 7 и 12 для переменных a и b получаем интервал индексов от 7 до 18 включительно.

При выполнении команды index=txt.LastIndexOf(symb, b, b+1) вычисляется индекс последнего (от начала текста) вхождения буквы, являющейся значением переменной symb, в текст из переменной

txt, причем поиск выполняется в направлении начала текста от позиции с индексом, определяемым значением переменной b (значение 12). В итоге переменная index получает значение 8 (такое же, как было до этого).

ПОДРОБНОСТИ

Если символ находится на позиции с индексом b, то от начала текста до этого символа всего находится b+1 символов. При выполнении инструкции txt.LastIndexOf(symb, b, b+1) текст из переменной txt просматривается на предмет наличия в нем символа, являющегося значением переменной symb. Текст просматривается, начиная с символа с индексом, определяемым значением второго аргумента b метода LastIndexOf(). Текст просматривается в направлении к началу текста, до первого совпадения. Первое совпадение с конца — это последнее совпадение с начала. Количество символов, которые просматриваются, определяется третьим аргументом метода LastIndexOf(). Значение этого аргумента b+1 (при значении второго аргумента b) означает, что просматриваются все символы до начала текста.

Затем переменной symb присваивается новое значение 'ы', и командой index=txt.IndexOf(symb) вычисляется индекс первого вхождения буквы 'ы' в текст из переменной txt. Но поскольку такой буквы в тексте нет, то переменная index получает значение -1.

Наконец, мы объявляем и инициализируем символьный массив s, состоящий из трех элементов (буквы 'ы', 'а' и 'д'). В результате выполнения команды index=txt.IndexOfAny(s) вычисляется индекс первого вхождения любой из перечисленных букв в текст txt. Буквы 'ы', как мы уже знаем, в тексте нет совсем. Буква 'а' в тексте из переменной txt впервые встречается в позиции с индексом 2. Буква 'д' первый раз встречается в позиции с индексом 6. То есть буква 'а' встречается раньше всех прочих букв из массива s. Следовательно, переменная index получает значение 2. Командой index=txt.IndexOfAny(s, index+1) вычисляется индекс второго вхождения в текст txt любой из букв из массива s. По факту выполняется поиск первого вхождения букв из массива s в текст txt, начиная с индекса index+1. Так получается, что это индекс 6.

При выполнении команды index=txt.LastIndexOfAny(s) вычисляется индекс последнего вхождения в текст из переменной txt букв из массива s. Получаем значение 38 (последнее вхождение буквы 'а').

Командой `index=txt.IndexOfAny(s, a, b)` вычисляется первое вхождение в текст `txt` букв из массива `s`, начиная с индекса, определяемого переменной `a` (значение `7`), и проверяются символы в тексте в количестве, определяемом значением переменной `b` (значение `12`). Другими словами, исследуется (на предмет наличия совпадений) интервал индексов от `7` до `18` включительно (от `a` до `a+b-1`). Получаем значение `8` для переменной `index`.

Команда `index=txt.LastIndexOfAny(s, a+b-1, b)` используется для вычисления последнего вхождения букв из массива `s` в текст из переменной `txt`, причем поиск начинается с элемента с индексом `a+b-1` (значение `18`) и выполняется в направлении к началу текста по `12` символам (значение переменной `b`). То есть речь идет все о том же диапазоне индексов от `7` до `18` включительно. Переменная `index` получает значение `17` (индекс последнего вхождения буквы `'a'` в указанном выше интервале).

НА ЗАМЕТКУ

Метод `Contains()` позволяет проверить, содержится ли подстрока в строке. Метод вызывается из объекта текста, в котором проверяется наличие подстроки. Подстрока передается аргументом методу. Результатом возвращается логическое значение (тип `bool`): значение `true` — если подстрока содержится в тексте, и значение `false` — если подстрока не содержится в тексте.

Методы `StartsWith()` и `EndsWith()` позволяют проверить, начинается или заканчивается текст определенной подстрокой. Подстрока передается аргументом, а соответствующий метод вызывается из текстового объекта, который проверяется на наличие в начале (метод `StartsWith()`) или конце (метод `EndsWith()`) подстроки. Результатом возвращается логическое значение, определяющее наличие или отсутствие подстроки в начале/конце текста.

Методы `Insert()`, `Remove()` и `Replace()` используются соответственно для вставки одной строки в другую, удаления части строки и замены одной части строки на другую.

НА ЗАМЕТКУ

Еще раз подчеркнем, что во всех перечисленных случаях речь идет не об изменении исходного текстового объекта, а о создании нового текстового объекта на основе уже существующего.

Методу `Insert()` при вызове передается первый целочисленный аргумент, определяющий место вставки подстроки, и собственно подстрока, предназначенная для вставки в исходную строку. Результатом метод возвращает новый созданный текстовый объект.

Методу `Remove()` передается два целочисленных аргумента: первый определяет позицию, начиная с которой в текстовой строке удаляются символы, а второй аргумент определяет количество удаляемых символов. Новая строка, сформированная путем удаления из исходного текста символов в указанном диапазоне индексов, возвращается результатом метода.

Методу `Replace()` аргументами могут передаваться два символа или две текстовые строки. При вызове метода из текстового объекта формируется (и возвращается результатом) новая текстовая строка, которая получается заменой в исходном тексте символа или подстроки, указанной первым аргументом, на символ или подстроку, указанную вторым аргументом.

Также стоит упомянуть метод `Substring()`, который позволяет извлечь подстроку из текста. Аргументом методу передается целочисленное значение, определяющее индекс, начиная с которого извлекается подстрока (подстрока извлекается до конца текста). Если передать второй целочисленный аргумент, то он будет определять количество символов, включаемых в извлекаемую подстроку. Подстрока возвращается результатом метода. Исходный текст при этом не меняется.

Как перечисленные выше методы используются на практике, иллюстрирует программа в листинге 7.5.

Листинг 7.5. Вставка и замещение текста

```
using System;
class ReplaceStringDemo{
   static void Main(){
      // Исходный текст:
      String txt="Мы изучаем C#";
      // Проверка исходного текста:
      Console.WriteLine(txt);
      // Текстовая переменная:
```

```
        String str;
        // Вставка слова в текст:
        str=txt.Insert(11,"язык ");
        // Проверка текста:
        Console.WriteLine(str);
        // Вставка в текст слова с последующим замещением
        // другого текстового блока:
        str=str.Insert(3,"не ").Replace("C#","Java");
        // Проверка текста:
        Console.WriteLine(str);
        // Замена пробелов на подчеркивания:
        str=str.Replace(' ','_');
        // Проверка текста:
        Console.WriteLine(str);
        // Извлечение подстроки:
        str=str.Substring(2,12);
        // Проверка текста:
        Console.WriteLine(str);
        // Извлечение подстроки:
        str=txt.Substring(3);
        // Проверка текста:
        Console.WriteLine(str);
        // Проверка исходного текста:
        Console.WriteLine(txt);
    }
}
```

Ниже представлен результат выполнения программы.

Результат выполнения программы (из листинга 7.5)

```
Мы изучаем C#
Мы изучаем язык C#
Мы не изучаем язык Java
```

```
Мы_не_изучаем_язык_Java
_не_изучаем_
изучаем C#
Мы изучаем C#
```

В программе объявляется текстовая переменная `txt` со значением `"Мы изучаем C#"`. Значение этой переменной выводится в консоль. Также объявляется текстовая переменная `str`. Первое значение этой переменной присваивается командой `str=txt.Insert(11,"язык ")`. Значением является текст, который получается добавлением строки `"язык "` в текстовое значение, на которое ссылается переменная `txt`. Строка вставляется, начиная с позиции с индексом `11`.

При выполнении команды `str=str.Insert(3,"не ").Replace("C#","Java")` переменная `str` получает новое текстовое значение, которое вычисляется следующим образом. Сначала на основе текущего (вычисленного на предыдущем шаге) значения переменой `str` путем вставки в позицию с индексом `3` строки `"не "` формируется новый текст. Ссылка на этот текст возвращается инструкцией `str.Insert(3,"не ")`. Другими словами, мы можем интерпретировать выражение `str.Insert(3,"не ")` как текстовый объект. Из этого текстового объекта вызывается метод `Replace()` с аргументами `"C#"` и `"Java"`. Это означает, что на основе текста, сформированного командой `str.Insert(3,"не ")`, создается новый текст: вместо строки `"C#"` вставляется строка `"Java"`. То, что получилось, присваивается значением переменной `str`.

Командой `str=str.Replace(' ','_')` в тексте, на который ссылается переменная `str`, пробелы (символ `' '`) заменяются на подчеркивания (символ `'_'`). Получающийся в результате текст присваивается новым значением переменной `str`.

Командой `str=str.Substring(2,12)` из текущего текстового значения, на которое ссылается переменная `str`, извлекается подстрока: текст формируется из `12` символов, начиная с позиции с индексом `2` (при этом исходная срока не меняется). Сформированное значение присваивается переменной `str`. Еще один пример извлечения подстроки дает команда `str=txt.Substring(3)`. В данном случае из текста, на который ссылается переменная `txt`, извлекается подстрока: начиная с позиции с индексом `2` и до конца текста. Подстрока присваивается значением переменной `str`. Значение переменной `txt` не меняется.

НА ЗАМЕТКУ

Метод `Trim()`, если он вызван без аргументов, позволяет удалить начальные и конечные пробелы в тексте, из которого он вызван (метод возвращает новую текстовую строку). Если методу передать `char`-аргументы (произвольное количество), то они определяют символы, которые удаляются из начала и конца текста.

Есть и другие полезные методы, предназначенные для работы с текстом. Скажем, с помощью метода `ToUpper()` на основе текстового объекта, из которого вызывается метод, формируется текстовая строка, состоящая из прописных (больших) букв исходного текста. Метод `ToLower()` возвращает строку, состоящую из строчных (маленьких) букв исходного текста. Можно сказать и проще: метод `ToUpper()` делает все буквы большими, а метод `ToLower()` делает все буквы маленькими. Нужно только помнить, что исходные текстовые объекты (из которых вызываются методы) не меняются, а речь идет о возвращаемом методами тексте.

Метод `ToCharArray()` результатом возвращает ссылку на символьный массив (тип `char[]`) из букв исходного текста (текстового объекта, из которого вызывается метод). Метод полезен в случаях, когда текстовое значение нужно разбить на отдельные символы. Похожая задача решается с помощью метода `Split()`. Метод результатом возвращает текстовый массив (тип `string[]`), элементами которого являются подстроки из исходного текста. Текстовый объект, из которого вызывается метод, разбивается на подстроки, причем символы, которые являются разделителями для определения мест разбивки исходного текста, передаются аргументами методу. Количество таких символьных аргументов произвольно. Если первым аргументом методу передать символьный массив (элементы которого играют роль разделителей при разбивке строки), а вторым аргументом указать целое число, то оно будет определять максимальное количество подстрок, на которые разбивается исходная текстовая строка. Если методу `Split()` аргументы не переданы, то по умолчанию разделителем считается пробел. Пример использования методов `ToUpper()`, `ToLower()`, `ToCharArray()` и `Split()` представлен в листинге 7.6.

Листинг 7.6. Разбиение строк и регистр символов

```
using System;
class SplittingStringDemo{
```

```
static void Main(){
   // Исходная текстовая строка:
   String txt="Дорогу осилит идущий";
   // Отображение исходного текста:
   Console.WriteLine(txt);
   // Текстовая строка:
   String str;
   // Текст с символами в верхнем регистре:
   str=txt.ToUpper();
   // Проверка текстового значения:
   Console.WriteLine(str);
   // Текст с символами в нижнем регистре:
   str=txt.ToLower();
   // Проверка текстового значения:
   Console.WriteLine(str);
   // Переменная для текстового массива:
   String[] words;
   // Разбивка текста на подстроки:
   words=txt.Split();
   // Отображение подстрок:
   for(int k=0; k<words.Length; k++){
      Console.WriteLine((k+1)+": "+words[k]);
   }
   Console.WriteLine();
   // Разбивка текста на подстроки:
   words=txt.Split('у','и');
   // Отображение подстрок:
   for(int k=0; k<words.Length; k++){
      Console.WriteLine((k+1)+": "+words[k]);
   }
   Console.WriteLine();
   // Разбивка текста на подстроки:
```

```
      words=txt.Split(new char[]{'у','и'},3);
      // Отображение подстрок:
      for(int k=0; k<words.Length; k++){
         Console.WriteLine((k+1)+": "+words[k]);
      }
      // Переменная для символьного массива:
      char[] symbs;
      // Массив из символов, формирующих текст:
      symbs=txt.ToCharArray();
      // Отображение символов из массива:
      for(int k=0; k<symbs.Length; k++){
         Console.Write(symbs[k]+" ");
      }
      Console.WriteLine();
   }
}
```

Результат программы такой.

Результат выполнения программы (из листинга 7.6)

```
Дорогу осилит идущий
ДОРОГУ ОСИЛИТ ИДУЩИЙ
дорогу осилит идущий
1: Дорогу
2: осилит
3: идущий

1: Дорог
2:  ос
3: л
4: т
5: д
6: щ
```

```
7: й


1: Дорог
2:  ос
3: лит идущий
Д о р о г у  о с и л и т  и д у щ и й
```

В программе для выполнения операций с текстом объявляется текстовая переменная `txt` со значением `"Дорогу осилит идущий"`. Данное текстовое значение отображается в консольном окне. Мы используем еще одну текстовую переменную `str`, значение которой присваивается командой `str=txt.ToUpper()`. В результате переменной `str` присваивается текст `"ДОРОГУ ОСИЛИТ ИДУЩИЙ"`, в котором все буквы в верхнем регистре (большие). После выполнения команды `str=txt.ToLower()` переменная `str` получает значением текст `"дорогу осилит идущий"`, состоящий из строчных (маленьких) букв.

Затем в программе объявляется переменная `words` для текстового одномерного массива. При выполнении команды `words=txt.Split()` текстовая строка, на которую ссылается переменная `txt`, разбивается на подстроки. Создается текстовый массив, количество элементов которого определяется количеством подстрок, на которые разбивается исходный текст. Подстроки становятся элементами массива. Ссылка на массив записывается в переменную `words`. Поскольку метод `Split()` вызывается без аргументов, то разделителем при разбивке текста на подстроки является пробел. При этом сами пробелы в подстроки не включаются. С помощью оператора цикла каждая подстрока отображается в отдельной строке в консольном окне. Еще один пример разбивки текста на строки дается командой `words=txt.Split('у','и')`. В этом случае исходный текст, на который ссылается переменная `txt`, разбивается на подстроки, и индикатором для разбивки на подстроки являются символы `'у'` и `'и'`(сами эти символы в подстроки не включены). Подстроки формируют текстовый массив, ссылка на который записывается в переменную `words`. Подстроки отображаются в консоли (некоторые подстроки содержат по одной букве).

При выполнении команды `words=txt.Split(new char[]{'у', 'и'},3)` исходный текст разбивается на подстроки, разделителями являются символы `'у'` и `'и'`, но количество подстрок не превышает

значение 3. То есть текст разбивается на подстроки, но как только их становится 3, то дальше текст на подстроки не разбивается.

ПОДРОБНОСТИ

Первым аргументом методу `Split()` передается анонимный символьный массив, который создается инструкцией `new char[]{'у','и'}`.

В программе объявляется переменная символьного массива `symbs`. Командой `symbs=txt.ToCharArray()` на основе текста `txt` создается символьный массив, и ссылка на него записывается в переменную `symbs`. Массив состоит из букв текста, на который ссылается переменная `txt`. Элементы массива отображаются в консольном окне через пробел.

НА ЗАМЕТКУ

При всех перечисленных выше операциях текст, на который ссылается переменная `txt`, не меняется.

ПОДРОБНОСТИ

Если метод `ToCharArray()` создает символьный массив, то метод `CopyTo()` позволяет скопировать текстовое значение посимвольно в уже существующий массив. Метод статический (вызывается из класса `String`) и не возвращает результат. Аргументами методу передаются:

- индекс позиции (целое число), начиная с которой в исходном тексте выполняется копирование;
- ссылка на символьный массив (тип `char[]`), в который выполняется копирование;
- индекс элемента (целое число), начиная с которого в массив копируются символы из текста;
- количество символов (целое число), которые копируются из текста в массив.

Метод `ToString()`

— Фамилия.
— Рюриковичи мы.

из к/ф «Иван Васильевич меняет профессию»

При создании класса в нем можно описать метод `ToString()`. Этот метод особый. Специфика его в том, что метод автоматически вызывается каждый раз, когда нужно преобразовать объект класса к текстовому формату. Представим себе ситуацию, что имеется некоторый класс и на основе этого класса создан объект. И вот этот объект передается аргументом методу `WriteLine()`. Если в классе описан метод `ToString()`, то этот метод автоматически будет вызван и текст, который метод вернет результатом, будет передан аргументом методу `WriteLine()` вместо ссылки на объект. Аналогичная ситуация имеет место, если в некотором выражении к объекту прибавляется текст (или к тексту прибавляется объект). Если так, то из объекта вызывается метод `ToString()`, а результат (текст), который возвращается методом, подставляется в выражение вместо ссылки на объект.

ПОДРОБНОСТИ

Если называть вещи своими именами, то здесь мы имеем дело с наследованием класса и переопределением метода. Этим темам посвящена отдельная глава книги. Пока что мы приведем лишь краткую справку, необходимую для понимания принципов реализации программного кода.

Наследование позволяет создавать новые (называются производными) классы на основе уже существующих (они называются базовыми). «Призом» в таком подходе является то, что открытые члены исходного класса (на основе которого создается новый класс) наследуются в создаваемом классе: хотя они в новом классе явно не описаны, но они там есть. Существует особый класс `Object` (из пространства имен `System`), который находится в вершине иерархии наследования. Любые классы, в том числе и описываемые нами в программе, являются прямыми или косвенными наследниками класса `Object`. В классе `Object` есть метод `ToString()`, поэтому, даже если мы в нашем классе такой метод не описываем, он там есть. Метод вызывается при необходимости преобразовать ссылку на объект к текстовому формату. Другое дело, что если явно метод `ToString()` в классе не описать, то преобразование выполняется

так: вместо ссылки на объект получаем название класса, на основе которого объект создан. Это «поведение» метода `ToString()` по умолчанию — как оно определено в классе `Object`. Мы его можем «переопределить». Описывая в нашем классе метод `ToString()`, мы как раз выполняем переопределение этого метода. Это проявление общего универсального механизма, который позволяет переопределять в производном классе методы, унаследованные из базового класса. При переопределении методов используется ключевое слово `override`.

Таким образом, у нас имеется возможность задать способ преобразования ссылок на объекты пользовательского класса к текстовому формату. Для этого в данном классе следует описать метод `ToString()`. Формальные требования, предъявляемые к описанию метода `ToString()`, следующие.

- Метод описывается с ключевым словом `public` (метод открытый).
- Метод описывается с ключевым словом `override`, что означает переопределение унаследованного метода. Причина в том, что метод `ToString()` описан в классе `Object`, который находится в вершине иерархии наследования. В пользовательском классе метод `ToString()` наследуется, но мы, описывая его в явном виде, задаем новый алгоритм выполнения этого метода. Ключевое слово `override` является признаком того, что речь идет именно о переопределении метода, а не об описании нового метода.
- Результатом метод возвращает текст (то есть значение типа `String`).
- Метод, как отмечалось, называется `ToString()`, и у него нет аргументов.

Если все перечисленные условия выполнены, то это означает, что мы переопределили в классе метод `ToString()` и, следовательно, объекты этого класса готовы для использования в различных выражениях в качестве текстовых «заменителей». Пример переопределения и использования метода `ToString()` приведен в листинге 7.7.

Листинг 7.7. Переопределение метода ToString()

```
using System;
// Класс с переопределением метода ToString():
class MyClass{
   // Целочисленное поле:
```

```
   public int num;
   // Символьное поле:
   public char symb;
   // Конструктор:
   public MyClass(int n, char s){
      // Присваивание значений полям:
      num=n;
      symb=s;
      // Отображение сообщения о создании объекта.
      // Неявно вызывается метод ToString():
      Console.WriteLine("Создан новый объект\n"+this);
   }
   // Переопределение метода ToString():
   public override String ToString(){
      // Локальная текстовая переменная:
      String txt="Числовое поле: "+num;
      txt+="\nСимвольное поле: "+symb;
      // Результат метода:
      return txt;
   }
}
// Класс с главным методом программы:
class ToStringDemo{
   // Главный метод:
   static void Main(){
      // Создание нового объекта:
      MyClass obj=new MyClass(100,'A');
      // Новые значения полей объекта:
      obj.num=200;
      obj.symb='B';
      Console.WriteLine("Новые значения полей объекта");
      // Отображение значений полей объекта.
```

```
      // Неявно вызывается метод ToString():
      Console.WriteLine(obj);
      // Разбивка текста с описанием объекта на подстроки.
      // Метод ToString() вызывается в явном виде:
      String[] str=obj.ToString().Split('\n');
      Console.WriteLine("Явный вызов метода ToString() ");
      // Отображение подстрок, содержащих значения
      // полей объекта:
      for(int k=0; k<str.Length; k++){
         Console.WriteLine("[* "+str[k]+" *]");
      }
   }
}
```

Эта программа дает такой результат.

Результат выполнения программы (из листинга 7.7)

```
Создан новый объект
Числовое поле: 100
Символьное поле: A
Новые значения полей объекта
Числовое поле: 200
Символьное поле: B
Явный вызов метода ToString()
[* Числовое поле: 200 *]
[* Символьное поле: B *]
```

В программе описан класс `MyClass`. У этого класса есть два открытых поля: целочисленное `num` и символьное `symb`. В классе переопределяется метод `ToString()`. Описывается метод следующим образом. В теле метода объявляется локальная текстовая переменная `txt`. Начальное значение переменной формируется как объединение текста `"Числовое поле: "` и значения поля `num`. Затем к текущему значению переменной `txt` дописывается текст `"\nСимвольное поле: "` и значение поля `symb`. Таким образом, переменная `txt` содержит текст со значениями

полей объекта (подразумевается объект, из которого будет вызываться метод `ToString()`). После того как значение переменной `txt` сформировано, командой `return txt` это значение возвращается результатом метода.

НА ЗАМЕТКУ

Текст, возвращаемый в качестве результата методом `ToString()`, содержит символ перехода к новой строке `'\n'`. Это обстоятельство впоследствии используется при разбиении текста на подстроки.

В классе также описан конструктор с двумя аргументами, которые определяют значения полей создаваемого объекта. Еще конструктор содержит команду `Console.WriteLine("Создан новый объект\n"+this)`. Этой командой в консольном окне отображается сообщение о создании нового объекта и значении полей этого объекта. Причем при выполнении команды вызывается метод `ToString()` (метод вызывается, хотя явной команды вызова этого метода нет). Дело в том, что аргументом методу `WriteLine()` передается выражение, представляющее собой сумму текстового литерала и ссылки `this`, обозначающей созданный объект. Таким образом, мы имеем дело с выражением, в котором к тексту прибавляется ссылка на объект. Это тот случай, когда вызывается метод `ToString()` и полученный в результате текст подставляется вместо ссылки `this`.

В главном методе программы командой `MyClass obj=new MyClass(100,'A')` создается объект. При его создании (вследствие вызова конструктора) отображается сообщение со значениями полей созданного объекта. Затем полям объекта присваиваются новые значения. Командой `Console.WriteLine(obj)` отображаются новые значения полей объекта `obj`. Здесь для преобразования объекта `obj` к текстовому формату вызывается метод `ToString()`. Но этот метод можно вызывать и в явном виде, как обычный метод. Пример такой ситуации дается инструкцией `obj.ToString().Split('\n')`. Здесь из объекта `obj` вызывается метод `ToString()`. Результатом является текстовая строка. Из объекта этой строки вызывается метод `Split()` с аргументом `'\n'`. В результате создается массив, состоящий из подстрок, на которые разбивается исходный текст (результат инструкции `obj.ToString()`). Индикатором для разбивки текста на подстроки является символ `'\n'`. Такой символ в тексте, возвращаемом методом `ToString()`, всего один, поэтому текст разбивается на две подстроки. Ссылка на массив с подстроками записывается в переменную `str`. С помощью оператора цикла значения массива `str` отображаются в консольном окне, заключенные в комбинацию из квадратных скобок и звездочек.

Резюме

Граф — добрый. Мухи не обидит. Сатрап — он и есть сатрап.

из к/ф «О бедном гусаре замолвите слово»

- Текстовые значения реализуются в виде объектов класса `String` из пространства имен `System`. Идентификатор `string` является псевдонимом для инструкции `System.String`. Текстовая переменная ссылается на объект, содержащий текстовое значение.
- Текстовые литералы реализуются в виде объектов класса `String`.
- Создать текстовый объект можно, присвоив текстовой переменной значением литерал. Также имеется возможность создать текстовый объект на основе символьного массива. Существуют и некоторые иные способы создания текстовых объектов.
- Размер текста (количество символов в тексте) можно узнать с помощью свойства `Length`, а для обращения к отдельным символам в тексте текстовый объект индексируется как массив. Индексация символов в тексте начинается с нуля.
- После создания текстового объекта его содержимое изменить нельзя. Изменения в текст вносятся путем создания новых текстовых объектов, а ссылка на вновь созданный объект записывается в текстовую переменную.
- Сравнивать текстовые строки на предмет совпадения или несовпадения можно с помощью операторов `==` и `!=`. Также для этой цели может использоваться метод `Equals()`.
- Существует много методов, предназначенных для выполнения различных операций с текстовыми значениями. В частности, методы позволяют: выполнять поиск символов и подстрок в строке, разбивать строку на подстроки, извлекать подстроку из текста, разбивать текст на символы, выполнять удаление подстрок, замену символов и подстрок и многое другое. При выполнении операций с текстом новая текстовая строка является результатом операции, а исходный текст при этом не меняется.
- Описав в классе метод `ToString()`, можно задать способ преобразования объектов этого класса к текстовому формату.

Задания для самостоятельной работы

> Мы месяц по галактике «маму» попоем — и плапета у пас в кармане.
>
> *из к/ф «Кин-дза-дза»*

1. Напишите программу, в которой есть статический метод. Аргументом методу передается текстовое значение. Результатом метод возвращает текст, в котором, по сравнению с текстом-аргументом, между символами вставлены пробелы.

2. Напишите программу, в которой есть статический метод, возвращающий результатом текстовое значение и получающий аргументом текст. Результат метода — это переданный аргументом текст, в котором слова следуют в обратном порядке. Словами считать блоки текста, разделенные пробелами.

3. Напишите программу со статическим методом для сравнения текстовых строк. Строки на предмет совпадения сравниваются посимвольно. Правило сравнения символов такое: два символа считаются одинаковыми, если их коды отличаются не больше, чем на единицу. Текстовые строки совпадают, если у них совпадают символы (в указанном выше смысле).

4. Напишите программу со статическим методом, который выполняет сравнение текстовых строк. Текстовые строки сравниваются следующим образом: для каждого текстового значения определяется набор разных букв, входящих в текст (при этом количество вхождений буквы в текст значения не имеет). Текстовые строки считаются равными, если они содержат одинаковые наборы букв.

5. Напишите программу со статическим методом, определяющим позиции, на которых в тексте находится определенный символ. Аргументами методу передаются текст и символ. Результатом метод возвращает целочисленный массив, значения элементов которого — это индексы позиций, на которых символ (второй аргумент) находится в тексте (первый аргумент). Если символ в тексте не встречается, то метод результатом возвращает массив из одного элемента, значение которого равно `-1`.

6. Напишите программу со статическим методом, аргументом которому передается текст, а результатом возвращается символьный массив, состоящий из букв (без учета пробелов и знаков препинания), из которых

состоит текст. Если буква несколько раз встречается в тексте, в массиве она представлена одним элементом. Буквы в массиве-результате должны быть отсортированы в алфавитном порядке.

7. Напишите программу со статическим методом, который эмулирует работу метода `Substring()`. Аргументами статическому методу передается текст и два целочисленных аргумента. Результатом метод возвращает текстовую строку, которая состоит из символов текста (первый аргумент), начиная с позиции с индексом, определяемым вторым аргументом метода. Третий аргумент статического метода определяет количество символов, которые включаются в подстроку.

8. Напишите программу с классом, у которого есть текстовое поле. Значение текстовому полю присваивается при создании объекта класса. Также в классе должен быть метод, позволяющий вставить подстроку в текст из текстового поля. Аргументами методу передается подстрока для вставки в текст, а также индекс позиции, начиная с которой выполняется вставка. Переопределить в классе метод `ToString()` так, чтобы он возвращал значением текст из текстового поля.

9. Напишите программу с классом, в котором есть текстовое поле и символьное поле. Значение полям присваивается при создании объекта класса. В классе должен быть метод, возвращающий результатом массив из текстовых строк. Такие строки получаются разбиением на подстроки значения текстового поля. Символ, являющийся индикатором для разбивки на подстроки, определяется значением символьного поля. Переопределить в классе метод `ToString()` так, чтобы он возвращал текст со значениями полей объекта и подстроки, на которые разбивается текст из текстового поля.

10. Напишите программу с классом, у которого есть поле, являющееся ссылкой на целочисленный массив. При создании объекта массив заполняется случайными числами. Переопределите в классе метод `ToString()` так, чтобы метод возвращал текстовую строку со значениями элементов массива. Также строка должна содержать информацию о количестве элементов массива и среднем значении для элементов массива (сумма значений элементов, деленная на количество элементов).

Глава 8
ПЕРЕГРУЗКА ОПЕРАТОРОВ

От пальца не прикуривают, врать не буду.
А искры из глаз летят.

из к/ф «Формула любви»

Эта глава посвящена перегрузке операторов. Далее мы будем обсуждать вопросы, связанные с тем, как действие операторов «доопределяется» для случаев, когда операндом в выражении является объект пользовательского класса. В частности, нас будут интересовать такие темы:

- общие принципы описания операторных методов;
- перегрузка арифметических операторов;
- особенности перегрузки операторов сравнения;
- перегрузка логических операторов;
- правила перегрузки операторов приведения типа.

Рассмотрим мы и другие смежные вопросы, связанные с перегрузкой операторов.

Операторные методы

Желтые штаны! Два раза «ку»!

из к/ф «Кин-дза-дза»

Как мы уже знаем, в языке C# есть много различных операторов, которые позволяют выполнять всевозможные операции со значениями разных типов. Например, мы можем с помощью оператора «плюс» + сложить два числа и получить в результате число. Мы можем «сложить» с помощью того же оператора два текстовых значения и получить в результате текст. Существуют и другие ситуации, когда мы можем

использовать оператор + (или какой-нибудь иной оператор). Все они предопределены: то есть случаи, когда мы можем использовать тот или иной оператор, определены заранее и зависят от типа операндов, участвующих в операции. Другими словами, имеется ограниченное количество ситуаций, в которых могут использоваться операторы языка C#, а результат выполнения операций зависит от типа операндов. Например, мы можем к тексту прибавить число, но не можем текст умножить на число. В этом смысле наши возможности ограничены и строго детерминированы. Но есть в языке C# очень мощный и эффективный механизм, который связан с *перегрузкой операторов*. Основная идея в том, что для объектов пользовательских классов (то есть классов, которые мы описываем в программе) доопределяется действие встроенных операторов языка C#. Скажем, описывая класс, возможно путем несложных манипуляций добиться того, что объекты этого класса можно будет складывать, как числа. Или, например, к объекту класса можно будет прибавить число, и так далее. Смысл каждой такой операции определяем мы, когда описываем класс.

Чтобы определить способ применения того или иного оператора к объектам класса, в классе описывается специальный метод, который называется *операторным*. Это метод, но немножко специфический. Название операторного метода состоит из ключевого слова `operator` и символа оператора, для которого определяется способ применения к объектам класса. Например, если мы хотим определить операцию сложения для объектов класса с помощью оператора +, то в классе нужно описать операторный метод с названием `operator+`. Если в классе описан операторный метод с названием `operator*`, то для объектов класса определена операция «умножения»: к таким объектам можно применять оператор умножения `*`. Ну и так далее.

> **НА ЗАМЕТКУ**
>
> Перегружать можно не все операторы. Доступные для перегрузки операторы и особенности перегрузки некоторых операторов обсуждаются в этой главе.

Но мало знать название операторного метода. Есть некоторые требования, которым должны соответствовать операторные методы. Вот они.

- Операторный метод описывается в классе, для объектов которого определяется действие соответствующего оператора.

- Операторный метод описывается как статический (то есть с ключевым словом `static`) и открытый (с ключевым словом `public`).
- Операторный метод должен возвращать результат (то есть идентификатором типа результата не может быть ключевое слово `void`). Результат операторного метода — это значение выражения, в которое входят соответствующий оператор и операнды, отождествляемые с аргументами операторного метода.
- Аргументы операторного метода отождествляются с операндами выражения, обрабатываемого операторным методом. Аргумент может быть один, если речь идет об операторном методе для унарного оператора, и аргументов может быть два, если речь идет об операторном методе для бинарного оператора. Как минимум один аргумент операторного метода должен быть объектом того класса, в котором операторный метод описан.

ПОДРОБНОСТИ

Допустим, имеется класс, в котором описан операторный метод с названием `operator+`. А еще есть выражение вида `A+B`, в котором хотя бы один из операндов `A` и `B` является объектом упомянутого класса. Тогда для вычисления результата выражения `A+B` вызывается операторный метод с названием `operator+`. При этом операнд `A` интерпретируется как первый аргумент операторного метода, а операнд `B` передается вторым аргументом операторному методу.

Выше речь шла о бинарном операторе `+`. У бинарного оператора два операнда (в выражении `A+B` операнды — это `A` и `B`). Но бывают операторы унарные, у которых один операнд. Примером унарного оператора может быть инкремент `++` или декремент `--`. Если обрабатывается выражение вида `A++` и `A` является объектом класса, в котором описан операторный метод с названием `operator++`, то именно этот метод будет вызван для обработки выражения `A++`. Поскольку операторный метод определен для унарного оператора, то у метода всего один аргумент. Этим аргументом методу будет передан операнд `A`.

В языке C# перегружаться могут многие операторы, но не все. Например, среди *бинарных* операторов для перегрузки доступны такие: `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<` и `>>`. Также можно перегружать следующие *унарные* операторы: `+`, `-`, `!`, `~`, `++`, `--`, `true` и `false`. Несложно заметить, что выше перечислены в основном арифметические и побитовые операторы.

ПОДРОБНОСТИ

Операторы «плюс» + и «минус» – упомянуты и среди бинарных, и среди унарных. Ошибки здесь нет, поскольку данные операторы могут использоваться и как унарные, и как бинарные. В выражениях вида `A+B` и `A-B` операторы используются как бинарные. Но еще операторы «плюс» и «минус» могут использоваться в следующем формате: `+A` или `-A`. Здесь речь идет об унарных операторах. С унарным оператором «минус» мы сталкиваемся каждый раз, когда записываем отрицательные числа: унарный оператор «минус» служит индикатором отрицательного числа. Унарный оператор «плюс» может использоваться как индикатор положительного числа. Но на практике он обычно не используется. Тем не менее, если мы имеем дело с объектами, то вполне можем использовать с этими объектами не только бинарные, но и унарные операторы «плюс» и «минус». Для этого достаточно в соответствующем классе описать операторные методы для указанных операторов.

Отдельного внимания заслуживают операторы `true` и `false`. Речь идет о том, что можно определить способ проверки объекта на предмет «истинности» или «ложности». Объекты, для которых определена такая проверка (перегружены операторы `true` и `false`), могут использоваться в качестве условий в условных операторах и операторах цикла. Операторы `true` и `false` перегружаются только в паре: или оба, или ни одного. Перегрузке этих операторов посвящен отдельный раздел в главе.

Перегружать можно и операторы сравнения, но такие операторы должны перегружаться парами: == и !=, < и >, <= и >=. Также в языке C# есть специальные *операторы приведения типа*, которые позволяют задавать закон преобразования объекта одного типа в объект другого типа. В пользовательском классе можно описать оператор явного приведения типа (`explicit`-форма) или неявного приведения типа (`implicit`-форма).

Не перегружаются сокращенные операторы присваивания, такие как +=, -=, *=, /=, %=, &=, |=, ^=, <<= и >>=. Но при этом мы можем так переопределить базовые операторы (имеются в виду бинарные операторы +, –, *, /,%, &, |, ^, << и >>), что удастся использовать и сокращенные формы оператора присваивания. Похожая ситуация имеет место с логическими операторами `&&` и `||` — они не перегружаются. Однако если «правильным образом» перегрузить операторы `&` и `|`, то в программном коде к операндам, являющимся объектами класса с перегруженными операторами `&` и `|`, можно будет применять и операторы `&&` и `||`.

НА ЗАМЕТКУ

Ситуацию с «правильной перегрузкой» операторов `&` и `|` (для возможности использования операторов `&&` и `||`) мы рассмотрим отдельно.

Есть операторы, которые не перегружаются. Среди них = (присваивание), `.` (оператор «точка»), `?:` (тернарный оператор), `->` (оператор «стрелка», используемый при работе с указателями на структуры), `=>` (оператор «стрелка», используемый при описании лямбда-выражений), `new` (оператор создания нового объекта) и ряд других. В том числе не перегружается оператор `[]` «квадратные скобки» (в общем случае это оператор, который позволяет индексировать объекты — то есть указывать в квадратных скобках после имени объекта индекс или индексы). Вместе с тем в языке C# в классах можно определять *индексаторы*. Через механизм создания индексаторов реализуется возможность индексировать объекты.

НА ЗАМЕТКУ

Перегрузка операторов подразумевает, что для одного бинарного оператора может описываться несколько операторных методов, которые отличаются типом аргументов и, соответственно, которые обрабатывают выражения (на основе данного оператора) с операндами разного типа.

Далее мы более подробно и на конкретных примерах рассмотрим способы перегрузки разных групп операторов.

Перегрузка арифметических и побитовых операторов

— Герой! А в корнеты, господип губернатор, он разжалован за дуэль.
— Дуэль был из-за дамы?
— Ну разумеется, сударыпя. Из-за двух!

из к/ф «О бедном гусаре замолвите слово»

Как иллюстрацию к перегрузке операторов мы рассмотрим небольшой и очень простой пример, в котором выполняется перегрузка оператора сложения для объектов пользовательского класса. Пример представлен в листинге 8.1.

Листинг 8.1. Знакомство с перегрузкой операторов

```
using System;
// Класс с перегрузкой оператора сложения:
class MyClass{
   // Целочисленное поле:
   public int number;
   // Конструктор с целочисленным аргументом:
   public MyClass(int n){
      // Присваивание значения полю:
      number=n;
   }
   // Операторный метод для перегрузки оператора сложения:
   public static int operator+(MyClass a, MyClass b){
      // Локальная целочисленная переменная:
      int m=a.number+b.number;
      // Результат метода:
      return m;
   }
}
// Класс с главным методом:
class OverloadingOperatorPlusDemo{
   // Главный метод:
   static void Main(){
      // Создание объектов класса MyClass:
      MyClass A=new MyClass(100);
      MyClass B=new MyClass(200);
      // Целочисленная переменная:
      int num;
      // Вычисление суммы объектов:
      num=A+B;
      // Отображение результата вычисления суммы объектов:
      Console.WriteLine("A+B="+num);
```

```
    }
}
```

Результат выполнения программы будет таким.

Результат выполнения программы (из листинга 8.1)

```
A+B=300
```

Проанализируем программный код примера. В первую очередь нас будет интересовать класс `MyClass`, поскольку в нем есть операторный метод, с помощью которого выполняется перегрузка оператора сложения для объектов класса.

У класса есть целочисленное поле `number`, а также конструктор с одним аргументом (который определяет значение поля). Но нас интересует программный код с описанием операторного метода `operator+()` (комментарии в программном коде удалены):

```
public static int operator+(MyClass a, MyClass b){
    int m=a.number+b.number;
    return m;
}
```

О чем свидетельствует этот программный код? Атрибуты `public` и `static` являются обязательными. Название метода `operator+` означает, что метод соответствует оператору «плюс» +. Аргументов у метода два — то есть речь идет о бинарной операции. Оба аргумента описаны как такие, что относятся к классу `MyClass`. Следовательно, операторный метод будет вызываться для вычисления значения выражений вида `A+B`, в котором оба операнда `A` и `B` являются объектами класса `MyClass`. Идентификатором типа результата для операторного метода указано ключевое слово `int`. Поэтому результатом выражения вида `A+B` будет целое число. Чтобы понять, какое именно, необходимо проанализировать команды в теле метода. Там их всего две. Сначала командой `m=a.number+b.number` вычисляется сумма значений полей складываемых объектов, и результат записывается в переменную `m`. Затем значение переменной `m` возвращается результатом метода. Все это означает, что если вычисляется выражение вида `A+B`, в котором операнды `A` и `B` являются объектами класса `MyClass`, то результатом такого выражения является сумма значений полей `number` складываемых объектов. В том, что это действительно так, убеждаемся при выполнении главного

метода. Там создаются два объекта A и B класса MyClass. Значения полей объектов равны 100 и 200 соответственно (значение поля определяется аргументом, переданным конструктору при создании объекта). При выполнении команды num=A+B целочисленной переменной num значением присваивается результат выражения A+B. Проверка показывает, что переменная num получает значение 300 (сумма чисел 100 и 200), как и должно быть.

Мы могли переопределить оператор сложения и по-другому, как в плане операндов (аргументов метода), так и в плане результата метода.

НА ЗАМЕТКУ

Способ, которым мы определили операторный метод в рассмотренном выше примере, подразумевает, что складываются два объекта класса MyClass. Но мы при этом не сможем вычислить, например, сумму объекта и числа или сумму числа и объекта. Чтобы такие операции стали «законными», необходимо описать соответствующие версии операторного метода.

Еще одна версия программы с перегрузкой оператора «плюс» представлена в листинге 8.2. На этот раз в классе MyClass представлено несколько операторных методов с названием operator+.

Листинг 8.2. Еще один способ перегрузки оператора «плюс»

```
using System;
// Класс с перегрузкой оператора "плюс":
class MyClass{
   // Целочисленное поле:
   public int number;
   // Конструктор с одним аргументом:
   public MyClass(int n){
      // Присваивание значения полю:
      number=n;
   }
   // Переопределение метода ToString():
   public override String ToString(){
      // Результат метода:
      return "Поле number: "+number;
```

```
    }
    // Операторный метод для вычисления суммы двух объектов:
    public static MyClass operator+(MyClass a, MyClass b){
        // Целочисленная локальная переменная:
        int m=a.number+b.number;
        // Создание объекта класса:
        MyClass t=new MyClass(m);
        // Результат метода:
        return t;
    }
    // Операторный метод для вычисления суммы объекта
    // и целого числа:
    public static MyClass operator+(MyClass a, int x){
        // Целочисленная локальная переменная:
        int m=a.number+x;
        // Результат метода:
        return new MyClass(m);
    }
    // Операторный метод для вычисления суммы целого
    // числа и объекта:
    public static MyClass operator+(int x, MyClass a){
        // Результат метода:
        return a+x;
    }
    // Операторный метод для унарного оператора "плюс":
    public static int operator+(MyClass a){
        // Результат метода:
        return a.number;
    }
}
// Класс с главным методом:
class MoreOverloadingPlusDemo{
```

```
   // Главный метод:
   static void Main(){
      // Создание объектов:
      MyClass A=new MyClass(100);
      MyClass B=new MyClass(200);
      // Вычисление суммы объектов:
      MyClass C=A+B;
      // Проверка результата:
      Console.WriteLine(A);
      Console.WriteLine(B);
      Console.WriteLine(C);
      // Вычисление суммы объекта и целого числа:
      C=A+1;
      // Проверка результата:
      Console.WriteLine(C);
      // Вычисление суммы целого числа и объекта:
      C=10+A;
      // Проверка результата:
      Console.WriteLine(C);
      Console.Write("Унарный оператор: ");
      // Используется унарный "плюс":
      Console.WriteLine(+C);
      int num=(+A)+(+B);
      // Проверка результата:
      Console.WriteLine("Сумма чисел: "+num);
   }
}
```

При выполнении программы получаем такой результат.

Результат выполнения программы (из листинга 8.2)

```
Поле number: 100
Поле number: 200
```

```
Поле number: 300
Поле number: 101
Поле number: 110
Унарный оператор: 110
Сумма чисел: 300
```

В этой программе (по сравнению с предыдущим примером) класс `MyClass` немного «расширился». Кроме целочисленного поля `number` и конструктора с одним аргументом, у класса `MyClass` теперь есть переопределенный метод `ToString()`. Результатом метод возвращает текст со значением поля `number` объекта класса. Но нас конечно же интересуют в первую очередь операторные методы. Их в классе четыре. Все они называются `operator+`. Отличаются методы типом и количеством аргументов, а также типом возвращаемого результата. Версия метода с двумя аргументами, являющимися объектами класса `MyClass`, результатом возвращает объект класса `MyClass`. То есть теперь, если мы будем складывать два объекта класса `MyClass`, то в результате получим новый объект класса `MyClass`.

НА ЗАМЕТКУ

В предыдущей версии программы из листинга 8.1 при сложении двух объектов класса `MyClass` результатом получали целое число.

В теле метода командой `m=a.number+b.number` вычисляется сумма полей `number` объектов `a` и `b`, переданных аргументами методу (то есть операндов выражения, которое будет обрабатываться этим операторным методом). После этого командой `MyClass t=new MyClass(m)` создается новый объект класса `MyClass`. Значение поля `number` этого объекта равно сумме значений полей объектов `a` и `b`. Созданный таким образом объект возвращается результатом операторного метода.

Еще одна версия операторного метода с названием `operator+` подразумевает, что аргументами (операндами) являются объект класса `MyClass` и целое число.

ПОДРОБНОСТИ

Порядок аргументов в операторном методе и, соответственно, порядок операндов в выражении с перегружаемым оператором имеет значение. Ситуации, когда к объекту прибавляется число и когда

к числу прибавляется объект, обрабатываются разными версиями операторного метода для оператора сложения. Поэтому в программе мы отдельно описываем версию операторного метода, в которой первый аргумент — объект, а второй — целое число, а в еще одной версии операторного метода первый объект — целое число, а второй аргумент — объект. Мы определяем результат сложения числа и объекта так же, как и результат сложения объекта и числа. Но в общем случае результат таких операций может быть различным.

Результатом выражения, в котором к объекту класса `MyClass` прибавляется целое число, является объект класса `MyClass`. Его поле `number` равно сумме значений поля `number` первого аргумента (объект `a`) и второго целочисленного аргумента (обозначен как `x`). Поэтому в теле метода командой `m=a.number+x` вычисляется сумма значения поля объекта и целого числа, а затем результатом метода возвращается объект, созданный инструкцией `new MyClass(m)`.

ПОДРОБНОСТИ

Инструкцией `new MyClass(m)` создается новый объект, но ссылка на него в объектную переменную не записывается. Такие объекты называют анонимными. В данном случае выражение `new MyClass(m)` указано после инструкции `return`. При выполнении инструкции `new MyClass(m)`, как отмечалось выше, создается новый объект. А результатом инструкции является ссылка на этот созданный объект. Поэтому получается, что операторный метод результатом возвращает ссылку на созданный объект, что нам и нужно.

В принципе, мы могли бы объявить объектную переменную класса `MyClass`, записать в нее ссылку на созданный объект, а затем указать данную объектную переменную в `return`-инструкции. Так мы поступили в версии операторного метода, в которой оба аргумента — объекты класса `MyClass`. В данном случае мы обошлись без вспомогательной объектной переменной.

Еще в программе описана версия операторного метода для оператора «плюс», в которой первый аргумент целочисленный (обозначен как `x`), а второй (обозначен как `a`) является объектом класса `MyClass`. В теле этой версии метода всего одна команда `return a+x`. И это примечательный момент. Результатом метода возвращается сумма объекта и целого числа. Выражение `a+x` имеет смысл, только если в классе описана версия операторного метода для оператора «плюс», в которой первый аргумент является объектом, а второй аргумент — целое число. Такая

версия метода в классе MyClass описана. Именно она вызывается для вычисления значения выражения a+x.

> **НА ЗАМЕТКУ**
>
> Образно выражаясь, операторный метод для вычисления суммы целого числа и объекта мы определяем так: при вычислении суммы целого числа и объекта нужно вычислить сумму этого объекта и целого числа. В этом смысле операция вычисления суммы целого числа и объекта является коммутативной: не имеет значения, в каком порядке указаны операнды. Но это не есть общее правило. Просто мы так определили правила сложения объектов и чисел. В принципе, мы могли определить их иначе, и результат сложения объекта и числа отличался бы (не только по значению, но и по типу результата) от результата сложения целого числа и объекта.

В классе MyClass есть версия операторного метода operator+() с одним аргументом (объект класса MyClass). Эта версия операторного метода перегружает унарный оператор «плюс». Результат выражения с унарным оператором «плюс» является целым числом — значением поля number объекта-операнда.

В главном методе программы проверяется функциональность описанных в классе MyClass операторных методов. Там создается два объекта (A и B) класса MyClass со значениями соответственно 100 и 200 для поля number. Объектная переменная C класса MyClass значение получает при выполнении команды C=A+B. То есть это сумма двух объектов. Результатом является новый объект, поле number которого равно 300 (сумма полей объектов A и B). При выполнении команды C=A+1 получаем новый объект (ссылка на который записывается в переменную C), и значение поля number этого объекта равно 101 (сумма поля number объекта A и второго числового операнда 1). Наконец, при выполнении команды C=10+A создается еще один объект со значением поля 110. Ссылка на объект записывается в переменную C. Значение поля вычисляется как сумма первого операнда 10 и значения поля number объекта A.

Значением выражения +C является целое число — это значение поля number объекта C. Здесь при вычислении выражения +C вызывается версия операторного метода operator+() для унарного оператора «плюс». Поэтому при выполнении команды Console.WriteLine(+C) в консольное окно выводится значение поля number объекта C. А вот при выполнении команды Console.WriteLine(C) в консоли

отображается текст, возвращаемый методом `ToString()` (то есть здесь речь идет о преобразовании объекта `C` к текстовому формату).

Аналогично, значением выражения `+A` есть значение поля `number` объекта `A`, а значением выражения `+B` есть значение поля `number` объекта `B`. Поэтому при выполнении команды `num=(+A)+(+B)` целочисленная переменная `num` получает значением сумму полей объектов `A` и `B`.

В завершение раздела рассмотрим еще один пример, в котором используется перегрузка разных арифметических и побитовых операторов. Программа представлена в листинге 8.3.

Листинг 8.3. Перегрузка арифметических и побитовых операторов

```
using System;
// Класс с перегрузкой арифметических и
// побитовых операторов:
class MyData{
   private int code;    // Закрытое целочисленное поле
   private char symb;   // Закрытое символьное поле
   private string text; // Закрытое текстовое поле
   // Конструктор с тремя аргументами:
   public MyData(int n, char s, string t){
      code=n;   // Значение целочисленного поля
      symb=s;   // Значение символьного поля
      text=t;   // Значение текстового поля
   }
   // Перегрузка метода ToString():
   public override string ToString(){
      // Локальная текстовая переменная:
      string txt="Поля объекта:\n";
      txt+="Числовое поле: "+code+"\n";
      txt+="Символьное поле: \'"+symb+"\'\n";
      txt+="Текстовое поле: \""+text+"\"\n";
      txt+="----------------------";
      // Результат метода:
```

```
    return txt;
}
// Операторный метод для вычисления суммы объекта
// и целого числа:
public static MyData operator+(MyData obj, int n){
    return new MyData(+obj+n,-obj,~obj);
}
// Операторный метод для вычисления разности объекта
// и целого числа:
public static MyData operator-(MyData obj, int n){
    return new MyData(+obj-n,-obj,~obj);
}
// Операторный метод для вычисления суммы целого
// числа и объекта:
public static int operator+(int n, MyData obj){
    return n+(+obj);
}
// Операторный метод для вычисления разности целого
// числа и объекта:
public static int operator-(int n, MyData obj){
    return n-(+obj);
}
// Операторный метод для вычисления суммы объекта
// и текстовой строки:
public static MyData operator+(MyData obj, string t){
    return new MyData(+obj,-obj, t);
}
// Операторный метод возвращает результатом текст:
public static string operator~(MyData obj){
    return obj.text;
}
// Операторный метод возвращает результатом число:
```

```
public static int operator+(MyData obj){
   return obj.code;
}
// Операторный метод возвращает результатом символ:
public static char operator-(MyData obj){
   return obj.symb;
}
// Операторный метод возвращает результатом
// символ из текста:
public static char operator>>(MyData obj, int k){
   return (~obj)[k];
}
// Операторный метод возвращает результатом
// символ из текста:
public static char operator<<(MyData obj, int k){
   return (~obj)[(~obj).Length-k-1];
}
// Операторный метод возвращает результатом объект:
public static MyData operator^(MyData a, MyData b){
   string txt=~a+" & "+~b;        // Локальная переменная
   return new MyData(+a,-b, txt); // Результат метода
}
// Операторный метод для увеличения значения
// целочисленного поля объекта:
public static MyData operator++(MyData obj){
   obj.code+=10;   // Увеличение значения поля
   return obj;     // Результат метода
}
// Операторный метод для уменьшения значения
// целочисленного поля объекта:
public static MyData operator--(MyData obj){
   obj.code-=10;   // Уменьшение значения поля
```

```
      return obj;     // Результат метода
   }
}
// Класс с главным методом:
class OverloadingOperatorsDemo{
   // Главный метод:
   static void Main(){
      // Создание объектов:
      MyData A=new MyData(100,'A',"Alpha");
      MyData B=new MyData(200,'B',"Bravo");
      // Новый объект:
      MyData C=A^B;
      // Проверка результата:
      Console.WriteLine(A);
      Console.WriteLine(B);
      Console.WriteLine(C);
      // Новый объект:
      C=B^A;
      // Проверка результата:
      Console.WriteLine(C);
      // Переменные:
      int n=+A;      // Значение поля code объекта A
      char s=-A;     // Значение поля symb объекта A
      string t=~A;   // Значение поля text объекта A
      // Проверка результата:
      Console.WriteLine("Объект A: поля {0}, \'{1}\' и \"{2}\"\n", n, s, t);
      // Увеличение значения поля code объекта A:
      A++;
      // Проверка результата:
      Console.WriteLine(A);
      // Увеличение значения поля code объекта A и
      // проверка результата:
```

```
Console.WriteLine(++A);
// Уменьшение значения поля code объекта B и
// проверка результата:
Console.WriteLine(B--);
// Уменьшение значения поля code объекта B:
--B;
// Проверка результата:
Console.WriteLine(B);
// К объекту прибавляется число:
C=A+1000;
// Проверка результата:
Console.WriteLine(C);
// Из объекта вычитается число:
C=A-100;
// Проверка результата:
Console.WriteLine(C);
// Сумма и разность числа и объекта:
Console.WriteLine("Сумма и разность: {0} и {1}\n",2000+A,1000-A);
// Прибавление к объекту текстовой строки:
C=A+"Charlie";
// Проверка результата:
Console.WriteLine(C);
// Посимвольное отображение значения текстового поля
// объекта C:
for(int k=0; k<(~C).Length; k++){
   Console.Write((C>>k)+" ");
}
Console.WriteLine();
// Символы из текстового поля объекта C
// отображаются в обратном порядке:
for(int k=0; k<(~C).Length; k++){
   Console.Write((C<<k)+" ");
}
```

```
      Console.WriteLine();
      // Прибавление объекта к текстовой строке:
      t="Объект C. "+C;
      // Проверка результата:
      Console.WriteLine(t);
   }
}
```

При выполнении этой программы получаем результат, представленный ниже.

Результат выполнения программы (из листинга 8.3)

```
Поля объекта:
Числовое поле: 100
Символьное поле: 'A'
Текстовое поле: "Alpha"
----------------------
Поля объекта:
Числовое поле: 200
Символьное поле: 'B'
Текстовое поле: "Bravo"
----------------------
Поля объекта:
Числовое поле: 100
Символьное поле: 'B'
Текстовое поле: "Alpha & Bravo"
----------------------
Поля объекта:
Числовое поле: 200
Символьное поле: 'A'
Текстовое поле: "Bravo & Alpha"
----------------------
Объект A: поля 100, 'A' и "Alpha"
```

```
Поля объекта:
Числовое поле: 110
Символьное поле: 'A'
Текстовое поле: "Alpha"
----------------------
Поля объекта:
Числовое поле: 120
Символьное поле: 'A'
Текстовое поле: "Alpha"
----------------------
Поля объекта:
Числовое поле: 190
Символьное поле: 'B'
Текстовое поле: "Bravo"
----------------------
Поля объекта:
Числовое поле: 180
Символьное поле: 'B'
Текстовое поле: "Bravo"
----------------------
Поля объекта:
Числовое поле: 1120
Символьное поле: 'A'
Текстовое поле: "Alpha"
----------------------
Поля объекта:
Числовое поле: 20
Символьное поле: 'A'
Текстовое поле: "Alpha"
----------------------
Сумма и разность: 2120 и 880
```

```
Поля объекта:
Числовое поле: 120
Символьное поле: 'A'
Текстовое поле: "Charlie"
----------------------
C h a r l i e
e i l r a h C
Объект C. Поля объекта:
Числовое поле: 120
Символьное поле: 'A'
Текстовое поле: "Charlie"
----------------------
```

Код достаточно большой, но одновременно и несложный. Мы прокомментируем лишь основные моменты. Основу кода составляет класс `MyData`. У класса есть три закрытых поля: целочисленное поле `code`, символьное поле `symb` и текстовое поле `text`. У класса всего один конструктор с тремя аргументами, которые определяют значения полей создаваемого объекта. Также в классе описан метод `ToString()`, возвращающий результатом текстовую строку с информацией о значении полей объекта. Для удобства восприятия информации текстовая строка содержит несколько инструкций `\n` (для отображения текста в новой строке консоли), а в конце отображается импровизированная линия из дефисов.

> **НА ЗАМЕТКУ**
>
> В программе в качестве идентификатора текстового типа использовано ключевое слово `string`. Напомним, что идентификатор `string` является псевдонимом для инструкции `System.String`. На практике для обозначения текстового типа обычно используют идентификатор `string`.

Для доступа к полям мы перегружаем унарные операторы `+`, `-` и `~`. Если `obj` — объект класса `MyData`, то значением выражения `+obj` является значение поля `code` объекта `obj`, значением выражения `-obj` является значение поля `symb` объекта `obj`, а значением выражения `~obj` является текстовое значение поля `text` объекта `obj`. Именно такие ссылки

на поля в основном используются в программном коде класса MyData. Кроме этого определяются такие операции с объектами класса MyData:

- Сумма объекта и целого числа: результатом является объект класса MyData. Значение поля code объекта-результата получается суммированием значения поля code объекта-операнда и числа, прибавляемого к объекту. Все остальные поля у объекта-результата такие же, как и у объекта-операнда.
- Разность объекта и целого числа: результатом является новый объект класса MyData. Целочисленное поле этого объекта вычисляется как разность поля code объекта-операнда и целого числа (второй операнд), вычитаемого из объекта.
- Сумма целого числа и объекта: результатом является целое число. Это сумма числового поля объекта и числового операнда.
- Разность целого числа и объекта: результатом является целое число, получающееся вычитанием из числа (первый операнд) значения числового поля объекта (второй операнд).
- Сумма объекта и текстовой строки: результатом является новый объект. У него числовое и символьное поле совпадает с соответствующими полями объекта-операнда, а значение текстового поля определяется прибавляемой текстовой строкой.
- Операторы инкремента ++ и декремента -- переопределяются так, что у объекта-операнда числовое поле соответственно увеличивается на 10 и уменьшается на 10. Причем результатом выражения на основе оператора инкремента/декремента является ссылка на объект-операнд.
- Бинарный оператор ^ переопределен таким образом, что результатом возвращается ссылка на новый объект класса MyData (при том, что оба операнда — тоже объекты данного класса). Значения полей объекта-результата вычисляются так. Значение числового поля совпадает со значением числового поля первого операнда. Значение символьного поля совпадает со значением символьного поля второго операнда. Значение текстового поля получается объединением значений текстовых полей операндов.
- Бинарные операторы >> и << перегружены таким образом, что результатом возвращается символ. Если obj — это объект класса MyData, а k — целое число, то результатом выражения obj>>k является символ с индексом k в текстовом поле объекта obj, а результатом

выражения obj<<k тоже является символ из текстового поля объекта obj, но отсчет ведется с конца текстового поля.

ПОДРОБНОСТИ

Текст из текстового поля объекта obj возвращается выражением ~obj. Символ с индексом k в этом текстовом значении вычисляется выражением (~obj)[k]. Круглые скобки нужны для изменения порядка применения оператора ~ и квадратных скобок.

Если под k подразумевать индекс, отсчитываемый с конца текстовой строки, то он соответствует «обычному» (отсчитываемому с начала строки) индексу (~obj).Length-k-1. Здесь следует учесть, что количество символов в тексте вычисляется выражением (~obj).Length. Полная ссылка на нужный символ выглядит как (~obj)[(~obj).Length-k-1].

В главном методе программы проверяется работа перегруженных операторов. Мы создаем два объекта A и B класса MyData, после чего выполняем некоторые операции с ними. На что стоит обратить внимание? Наиболее интересные позиции.

- Результат выполнения команд C=A^B и C=B^A разный. То есть в выражении на основе оператора ^ имеет значение, какой операнд первый, а какой — второй.
- С помощью унарных операторов +, – и ~ можно получить значения полей объекта. Например, для объекта A выражение +A — это значение поля code, выражение -A — это значение поля symb, а выражение ~A — это значение поля text.
- Операторы инкремента и декремента можно использовать в префиксной (++A и --B) и постфиксной (A++ и B--) формах. Выражение на основе оператора инкремента или декремента (например, ++A или B--) является ссылкой на объект-операнд. Скажем, значение выражения ++A или A++ — это ссылка на объект A.
- Прибавляя к объекту число или вычитая из объекта число, получаем объект (например, команды C=A+1000 и C=A-100). А вот прибавляя к числу объект и вычитая из числа объект, получаем число (например, команды 2000+A и 1000-A).
- Узнать длину текста в текстовом поле, например объекта C, можно с помощью инструкции (~C).Length. Инструкция C>>k дает значение символа с индексом k, а инструкция C<<k дает значение

символа с индексом k, если текстовую строку индексировать с конца в начало. Для изменения порядка выполнения операторов используются круглые скобки.

- Если к объекту прибавляется текст, как в команде `C=A+"Charlie"`, то результатом получаем новый объект. Такая операция обрабатывается с помощью соответствующего операторного метода. Если же к тексту прибавляется объект, как в команде `t="Объект C. "+C`, то для такого случая специальный операторный метод не предусмотрен. Поэтому объект `C` вызовом метода `ToString()` преобразуется к текстовому формату, происходит объединение строк и результат присваивается текстовой переменной `t`.

НА ЗАМЕТКУ

Если оператор в принципе перегружаем, то выполнять перегрузку можно по-разному. Другими словами, конкретный способ перегрузки оператора определяется решаемой задачей и фантазией программиста. Однако есть общая рекомендация, состоящая в том, что перегрузку операторов желательно выполнять с учетом «основного» назначения оператора. Например, если перегружается бинарный оператор «плюс» для объектов определенного класса, то желательно, чтобы определяемая при перегрузке операция могла интерпретироваться в некотором смысле как сложение объектов. Но это лишь рекомендация, не более того.

Перегрузка операторов сравнения

> Девочка, вы тут самые умные? Это вам кто-нибудь сказал, или вы сами решили?
>
> *из к/ф «Кин-дза-дза»*

Операторов сравнения всего шесть, и, как отмечалось выше, перегружаются они парами. Правила такие:

- Если перегружен оператор «меньше» <, то должен быть перегружен и оператор «больше» >, и наоборот.
- Операторы «меньше или равно» <= и «больше или равно» >= перегружаются в паре.
- В паре перегружаются операторы «равно» == и «не равно» !=. При перегрузке этих операторов обычно также переопределяют методы

`Equals()` и `GetHashCode()`, наследуемые из класса `Object`. Этот вопрос мы обсудим более детально, когда будем рассматривать перегрузку операторов «равно» и «не равно».

В своих «оригинальных» версиях операторы сравнения возвращают результатом значения логического типа. Обычно операторы сравнения для объектов пользовательских классов перегружаются так, чтобы результат был логического типа. Но это не есть обязательное условие перегрузки операторов сравнения. В принципе, результат операторного метода для операторов сравнения может быть любого типа, не обязательно логического.

Также важно понимать, что подразумевается под «парной» перегрузкой операторов сравнения. Дело в том, что для одного и того же оператора может быть описано несколько версий операторного метода, которые отличаются типом аргументов и/или типом результата. То, что операторы сравнения перегружаются парами, означает, что если описана версия операторного метода для оператора сравнения с аргументами и результатом определенного типа, то должна быть описана версия операторного метода для другого оператора из пары с аргументами и результатом такого же типа. Например, если мы описали в классе операторный метод для оператора >= с результатом логического типа и аргументами, которые оба являются объектами класса, то в классе должен быть описан и операторный метод для оператора <= с результатом логического типа и аргументами, которые являются объектами класса.

Небольшой пример перегрузки операторов сравнения <= и >= представлен в листинге 8.4.

Листинг 8.4. Знакомство с перегрузкой операторов сравнения

```
using System;
// Класс с перегрузкой операторов сравнения:
class MyClass{
   // Символьное поле:
   public char symb;
   // Конструктор с одним аргументом:
   public MyClass(char s){
      // Присваивание значения полю:
      symb=s;
```

```
   }
   // Перегрузка оператора "меньше или равно":
   public static bool operator<=(MyClass a, MyClass b){
      if(a.symb<=b.symb) return true;
      else return false;
   }
   // Перегрузка оператора "больше или равно":
   public static bool operator>=(MyClass a, MyClass b){
      if(a.symb>=b.symb) return true;
      else return false;
   }
}
// Класс с главным методом:
class OverloadingCompOperatorsDemo{
   // Главный метод:
   static void Main(){
      // Создание объектов:
      MyClass A=new MyClass('A');
      MyClass B=new MyClass('B');
      MyClass C=new MyClass('A');
      // Использование операторов сравнения:
      Console.WriteLine("A<=B дает {0}", A<=B);
      Console.WriteLine("A>=B дает {0}", A>=B);
      Console.WriteLine("A<=C дает {0}", A<=C);
      Console.WriteLine("A>=C дает {0}", A>=C);
   }
}
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 8.4)

```
A<=B дает True
A>=B дает False
```

```
A<=C дает True
A>=C дает True
```

Мы в программе описали класс `MyClass` с символьным полем `symb`, конструктором с одним аргументом, а еще в классе описаны два операторных метода для операторов <= и >=. Подразумевается, что с помощью операторов сравниваются объекты класса `MyClass`. Методы возвращают логическое значение `true` или `false` в зависимости от результатов сравнения значений символьных полей объектов. В главном методе создаются три объекта, и для их сравнения используются операторы >= и <=. Ситуация достаточно простая. Поэтому хочется верить, что более подробных пояснений код программы и результат ее выполнения не требуют.

Еще один пример, более сложный, связанный с перегрузкой операторов сравнения (на этот раз перегружаются операторы <, <=, > и >=), представлен в листинге 8.5. Там, кроме прочего, встречается ситуация, когда оператор сравнения результатом возвращает значение, тип которого отличается от логического.

Листинг 8.5. Перегрузка нескольких операторов сравнения

```
using System;
// Класс с перегруженными операторами сравнения:
class MyClass{
   // Целочисленное поле:
   public int code;
   // Конструктор с одним аргументом:
   public MyClass(int n){
      // Присваивание значения полю:
      code=n;
   }
   // Перегрузка оператора "меньше или равно":
   public static MyClass operator<=(MyClass a, MyClass b){
      if(a.code<=b.code) return a;
      else return b;
   }
```

```
   // Перегрузка оператора "больше или равно":
   public static MyClass operator>=(MyClass a, MyClass b){
      if(a.code>=b.code) return a;
      else return b;
   }
   // Перегрузка оператора "меньше или равно":
   public static bool operator<=(MyClass a, int x){
      if(a.code<=x-1) return true;
      else return false;
   }
   // Перегрузка оператора "больше или равно":
   public static bool operator>=(MyClass a, int x){
      if(a.code>=x+1) return true;
      else return false;
   }
   // Перегрузка оператора "меньше":
   public static bool operator<(MyClass a, MyClass b){
      return a.code<b.code;
   }
   // Перегрузка оператора "больше":
   public static bool operator>(MyClass a, MyClass b){
      return a.code>b.code;
   }
   // Перегрузка оператора "меньше":
   public static int operator<(MyClass a, int x){
      return x-a.code;
   }
   // Перегрузка оператора "больше":
   public static int operator>(MyClass a, int x){
      return a.code-x;
   }
}
```

```
// Класс с главным методом:
class MoreOverloadingCompsDemo{
   // Главный метод:
   static void Main(){
      // Создание объектов:
      MyClass A=new MyClass(100);
      MyClass B=new MyClass(200);
      // Проверка результата:
      Console.WriteLine("Объект A: {0}", A.code);
      Console.WriteLine("Объект B: {0}", B.code);
      // Использование операторов "меньше" и "больше":
      Console.WriteLine("A<B дает {0}", A<B);
      Console.WriteLine("A>B дает {0}", A>B);
      // Объектная переменная:
      MyClass C;
      // Использование оператора "больше или равно":
      C=A>=B;
      // Новое значение поля:
      C.code=300;
      // Проверка результата:
      Console.WriteLine("Объект B: {0}", B.code);
      // Использование оператора "меньше или равно":
      C=A<=B;
      // Новое значение поля:
      C.code=150;
      // Проверка результата:
      Console.WriteLine("Объект A: {0}", A.code);
      // Новое значение поля:
      (B<=A).code=500;
      // Проверка результата:
      Console.WriteLine("Объект A: {0}", A.code);
      // Целочисленные переменные:
```

```
        int x=400, y=500, z=600;
        // Использование операторов "меньше или равно" и
        // "больше или равно":
        Console.WriteLine("A<={0} дает {1}", x, A<=x);
        Console.WriteLine("A>={0} дает {1}", x, A>=x);
        Console.WriteLine("A<={0} дает {1}", y, A<=y);
        Console.WriteLine("A>={0} дает {1}", y, A>=y);
        Console.WriteLine("A<={0} дает {1}", z, A<=z);
        Console.WriteLine("A>={0} дает {1}", z, A>=z);
        // Использование операторов "меньше" и "больше":
        Console.WriteLine("A<{0} дает {1}", z, A<z);
        Console.WriteLine("A>{0} дает {1}", x, A>x);
    }
}
```

На этот раз при выполнении программы получаем следующее.

Результат выполнения программы (из листинга 8.5)

```
Объект A: 100
Объект B: 200
A<B дает True
A>B дает False
Объект B: 300
Объект A: 150
Объект A: 500
A<=400 дает False
A>=400 дает True
A<=500 дает False
A>=500 дает False
A<=600 дает True
A>=600 дает False
A<600 дает 100
A>400 дает 100
```

Операторные методы перегружаются для класса MyClass. У класса есть целочисленное поле code и конструктор с одним аргументом (значение поля). В классе перегружаются такие операторы:

- Операторы <= и >= в случае, если оба операнда — объекты класса MyClass, результатом возвращают ссылку на объект класса MyClass. Мало того, это ссылка на один из двух объектов-операндов. Например, если A и B являются объектами класса MyClass, то результатом выражения A<=B будет ссылка на объект A, если значение поля code этого объекта не превышает значения поля code объекта B. Если значение поля code объекта A больше значения поля code объекта B, то результатом выражения A<=B является ссылка на объект B. Аналогично, результатом выражения A>=B является ссылка на объект A, если его поле code больше или равно значению поля code объекта B. Если значение поля code объекта A меньше значения поля code объекта B, то результатом выражения A>=B является ссылка на объект B.
- Операторы <= и >= можно использовать и в случае, если первый операнд является объектом класса MyClass, а второй операнд является целым числом. Результат операции с такими операндами является логическим значением true или false. Допустим, A является объектом класса MyClass, а через x обозначим некоторое целочисленное значение. Результат выражения A<=x будет равен true, если значение поля code объекта A меньше или равно числу x-1. В противном случае результат выражения A<=x равен false. А вот результат выражения A>=x равен true, если значение поля code объекта A больше или равно числу x+1. Иначе значение выражения A>=x равно false. Несложно заметить, что операторы <= и >= в данном случае перегружены так, что если значение поля code объекта A равно x, то значение выражения A<=x равно false, но и значение выражения A>=x также равно false. Это «поведение» операторов <= и >= не очень согласуется с нашей повседневной логикой, взращенной на сравнении числовых значений. Вместе с тем, перегружая операторы, мы можем добиться и такого «экзотического» поведения. Главное, чтобы в этом была необходимость.
- Операторы < и > для случая, когда сравниваются объекты класса MyClass, перегружаются очевидным образом: операторным методом возвращается результат сравнения (с помощью соответствующего оператора) значений полей code объектов-операндов.

- Операторы < и > также перегружаются для случая, когда первый операнд является объектом класса `MyClass`, а второй операнд является целым числом. В этом случае результатом операции сравнения возвращается целое число. Для оператора < оно вычисляется как разность значений числового операнда и значения поля `code` первого операнда-объекта. Для оператора > результат вычисляется как разность поля `code` первого объекта-операнда и числового значения, указанного вторым операндом.

В главном методе программы мы проверяем функциональность перегруженных операторов. Для этого создаем два объекта `A` и `B` класса `MyClass` со значениями полей `100` и `200` соответственно. Затем выполняются различные операции с использованием операторов <, >, <= и >=. При вычислении выражений `A<B` и `A>B` результатом является `true` и `false` в зависимости от значений полей сравниваемых объектов.

НА ЗАМЕТКУ

Логические значения `true` и `false` при выводе в консольное окно отображаются с большой буквы.

Мы объявляем объектную переменную `C` класса `MyClass`. При выполнении команды `C=A>=B` в переменную `C` записывается ссылка на тот же объект, на который ссылается переменная `B`. Поэтому когда выполняется команда `C.code=300`, то в действительности значение присваивается полю `code` объекта `B`. А вот при выполнении команды `C=A<=B` в переменную `C` записывается ссылка на объект `A`. Командой `C.code=150` присваивается новое значение полю `code` объекта `A`.

Значением выражения `B<=A` является ссылка на объект `A`. Поэтому команда `(B<=A).code=500` означает, что полю `code` объекта `A` присваивается значение `500`.

Также мы используем три целочисленные переменные `x`, `y` и `z` со значениями `400`, `500` и `600` соответственно. Эти переменные сравниваются с объектом `A`. Использование для сравнения операторов <= и >= дает в результате логическое значение. Достоин внимания результат сравнения объекта `A` с переменной `y` (команды `A<=y` и `A>=y`). Здесь имеет место ситуация, упомянутая ранее: значение поля `code` объекта `A` и значение переменной `y` совпадают. Поэтому оба выражения `A<=y` и `A>=y` значением дают `false`.

Сравнение объекта A с целочисленными переменными с помощью операторов > и < (команды A<z и A>x) результатом дает целое число (для оператора > это разность поля code объекта и числа, а для оператора < это разность числа и поля code объекта).

Похожим образом обстоят дела с перегрузкой операторов «равно» == и «не равно» !=. Правда, есть и небольшие особенности. Чтобы понять их суть, необходимо сделать небольшое отступление.

Даже если мы не перегружаем операторы == и != объектов класса, мы все равно можем использовать эти операторы (в отличие от операторов <, >, <= и >=) для сравнения объектов. Например, если объекты A и B относятся к одному и тому же классу, то операции вида A==B и A!=B вполне законны и имеют смысл. По умолчанию, если операторы == и != не перегружались, при выполнении команды A==B проверяются на предмет совпадения значения объектных переменных A и B. А в эти переменные фактически записаны адреса объектов, на которые переменные ссылаются. Поэтому результатом выражения A==B является значение true, если переменные A и B ссылаются на один и тот же объект. Если переменные ссылаются на физически разные объекты, результат выражения A==B равен false. При выполнении инструкции A!=B все происходит аналогичным образом, но проверка выполняется на предмет неравенства — то есть результат выражения A!=B равен true, если переменные A и B ссылаются на разные объекты, а если они ссылаются на один и тот же объект, то результат выражения A!=B равен false. Если нас такое «поведение» операторов == и != не устраивает, то мы при описании класса можем эти операторы перегрузить. Особых ограничений (за исключением того, что операторы == и != перегружаются в паре) здесь нет, в том числе и касательно типа результата, возвращаемого соответствующими операторными методами. Исключительно простой пример перегрузки операторов сравнения представлен в листинге 8.6.

Листинг 8.6. Перегрузка операторов «равно» и «не равно»

```
using System;
// Класс без перегрузки операторов сравнения:
class Alpha{
   // Целочисленное поле:
   public int code;
   // Конструктор с одним аргументом:
   public Alpha(int n){
```

```
      // Присваивание значения полю:
      code=n;
   }
}
// Класс с перегрузкой операторов сравнения:
class Bravo{
   // Целочисленное поле:
   public int code;
   // Конструктор с одним аргументом:
   public Bravo(int n){
      // Присваивание значения полю:
      code=n;
   }
   // Перегрузка оператора "равно":
   public static bool operator==(Bravo a, Bravo b){
      return a.code==b.code;
   }
   // Перегрузка оператора "не равно":
   public static bool operator!=(Bravo a, Bravo b){
      return !(a==b);
   }
}
// Класс с главным методом:
class OverloadingEqvOperatorDemo{
   // Главный метод:
   static void Main(){
      // Создание объектов класса Alpha:
      Alpha A1=new Alpha(100);
      Alpha A2=new Alpha(100);
      Alpha A3=new Alpha(200);
      // Проверка объектов на предмет
      // равенства или неравенства:
```

```
        Console.WriteLine("A1==A2 дает {0}", A1==A2);
        Console.WriteLine("A1!=A2 дает {0}", A1!=A2);
        Console.WriteLine("A1==A3 дает {0}", A1==A3);
        Console.WriteLine("A1!=A3 дает {0}", A1!=A3);
        // Создание объектов класса Bravo:
        Bravo B1=new Bravo(100);
        Bravo B2=new Bravo(100);
        Bravo B3=new Bravo(200);
        // Проверка объектов на предмет
        // равенства или неравенства:
        Console.WriteLine("B1==B2 дает {0}", B1==B2);
        Console.WriteLine("B1!=B2 дает {0}", B1!=B2);
        Console.WriteLine("B1==B3 дает {0}", B1==B3);
        Console.WriteLine("B1!=B3 дает {0}", B1!=B3);
    }
}
```

В принципе, эта программа компилируется, но с предупреждением. Предупреждение — это не ошибка. Это скорее признак непоследовательного подхода к составлению программного кода. Суть предупреждения в данном случае в том, что в программе перегружены операторы == и !=, но не переопределены методы `Equals()` и `GetHashCode()`, наследуемые из класса `Object`. Методы обсудим далее. Сначала проанализируем код программы и результат ее выполнения.

Результат выполнения программы (из листинга 8.6)

```
A1==A2 дает False
A1!=A2 дает True
A1==A3 дает False
A1!=A3 дает True
B1==B2 дает True
B1!=B2 дает False
B1==B3 дает False
B1!=B3 дает True
```

В программе описаны классы Alpha и Bravo. В каждом из классов есть целочисленное поле code и конструктор с одним аргументом. Принципиальное отличие между классами в том, что в классе Alpha операторы == и != не перегружаются, а в классе Bravo данные операторы перегружены. Перегружены они следующим образом. Операторный метод для оператора == описан с двумя аргументами класса Bravo, которые обозначены как a и b. Результатом метода возвращается значение выражения a.code==b.code. В этом выражении используется оператор сравнения ==, но здесь он применяется по отношению к значениям целочисленного типа (поля code объявлены как целочисленные), и поэтому в данном случае действие оператора == предопределено характеристиками языка C#. Результат выражения a.code==b.code равен true, если значения полей code объектов a и b совпадают. Иначе значение выражения a.code==b.code равно false. Таким образом, результат операторного метода для оператора == равен true, если у сравниваемых объектов значения полей code совпадают (при этом сами объекты могут быть физически разными). Если поля code имеют разные значения, операторный метод для оператора == возвращает значение false.

Операторный метод для оператора сравнения != определен так: для аргументов a и b (объекты класса Bravo) результатом метода возвращается значение выражения !(a==b). Здесь вычисляется значение выражения a==b. Поскольку a и b — объекты класса Bravo, то для вычисления выражения a==b используется операторный метод для оператора ==. Результатом является логическое значение. Затем с помощью оператора логического отрицания ! результат выражения a==b заменяется на противоположный. Следовательно, оператор != работает по такой схеме. Если для двух указанных операндов оператор == возвращает значение true, то оператор != для тех же операндов возвращает значение false. И наоборот, если оператор == возвращает значение false, то оператор != с теми же операндами вернет значение true.

В главном методе программы создается по три объекта класса Alpha и Bravo. Эти объекты сравниваются с помощью операторов == и !=. Интерес представляет результат сравнения объектов A1 и A2 класса Alpha, с одной стороны, и объектов B1 и B2 класса Bravo — с другой. Примечательны объекты тем, что у них одинаковые значения полей. При этом результатом сравнения объектов A1 и A2 на предмет равенства (выражение A1==A2) является значение false, а при сравнении на предмет

равенства объектов B1 и B2 получаем значение true. Причина в том, что при вычислении значения выражения A1==A2 (поскольку в классе Alpha операторы сравнения не перегружались) сравниваются ссылки на объекты (а они разные). А при вычислении значения выражения B1==B2, в соответствии с перегрузкой операторов сравнения в классе Bravo, сравниваются не ссылки, а значения полей объектов.

Теперь вернемся к методам Equals() и GetHashCode(), которые рекомендуется переопределять при перегрузке операторов == и !=. И сначала несколько слов о месте и роли этих методов.

Метод Equals() из класса Object предназначен для сравнения двух объектов на предмет равенства. Метод вызывается из одного объекта, а другой объект передается аргументом методу. При этом объекты могут относиться к разным классам. Сравнение объектов (того, из которого вызывается метод, и того, который передан аргументом методу) на предмет равенства выполняется на уровне сравнения ссылок на объекты. Метод Equals() возвращает значение true, если обе объектные переменные ссылаются на один и тот же объект. Если переменные ссылаются на разные объекты, метод Equals() возвращает значение false.

ПОДРОБНОСТИ

Метод Equals() описан в классе Object, и его аргумент объявлен как относящийся классу Object. Класс Object находится в вершине иерархии наследования, поэтому все классы, включая и описываемые нами в программе, неявно «получают в наследство» методы из класса Object. Это касается и метода Equals(). Мы можем вызывать этот метод из объекта описанного нами класса, даже если мы в классе такой метод не описали. Аргументом методу мы можем передавать объект любого класса, поскольку аргумент метода Equals() относится к классу Object, а объектная переменная класса Object может ссылаться на объект любого класса.

Метод GetHashCode() также описан в классе Object, поэтому наследуется во всех классах. У метода нет аргументов, а результатом метод возвращает целое число. Это целое число называется *хэш-кодом* объекта, из которого вызывается метод. Сам по себе хэш-код особого смысла не имеет. Хэш-коды используются для сравнения двух объектов на предмет равенства. Главное правило такое: если два объекта считаются одинаковыми (равными), то у них должны быть одинаковые хэш-коды.

НА ЗАМЕТКУ

Если два объекта равны, то их хэш-коды должны совпадать. Но если у двух объектов совпадают хэш-коды, то это не означает равенства объектов.

Таким образом, каждый раз при описании класса, вне зависимости от нашего желания, в этом классе будут «неявно присутствовать» методы `Equals()` и `GetHashCode()` — мы можем вызвать эти методы из объекта класса, несмотря на то что лично мы методы в классе и не описывали. По умолчанию эти методы работают по определенным алгоритмам, но важно то, что анализируются ссылки для сравниваемых объектных переменных. По умолчанию операторы == и != реализуются в строгом соответствии с таким подходом. То есть мы имеем дело с «великолепной четверкой» (методы `Equals()` и `GetHashCode()` и операторы == и !=), которая функционирует в согласованном, «гармонизированном» режиме. И как только мы перегружаем операторы сравнения == и !=, меняя способ определения «равенства» объектов, то эта «гармония» нарушается. Поэтому, перегружая операторы == и !=, желательно переопределить и методы `Equals()` и `GetHashCode()`. Как это можно было бы сделать, показано далее.

НА ЗАМЕТКУ

Переопределение методов — механизм, связанный с наследованием (наследование и перегрузка обсуждаются в одной из следующих глав). Методы `Equals()` и `GetHashCode()` наследуются в пользовательском классе, и мы их можем переопределить (описать новые версии этих методов). При переопределении методы описываются с ключевым словом `override`.

В листинге 8.7 вниманию читателя представлена программа, в которой наряду с перегрузкой операторов == и != еще и переопределяются методы `Equals()` и `GetHashCode()`.

НА ЗАМЕТКУ

Сразу отметим, что мы упростили ситуацию настолько, насколько это возможно. Цель была в том, чтобы проиллюстрировать общий подход.

Листинг 8.7. Перегрузка операторов сравнения и переопределение методов

```
using System;
// Класс с перегрузкой операторов сравнения == и !=
// и переопределением методов GetHashCode() и Equals():
class MyClass{
   // Целочисленное поле:
   public int code;
   // Символьное поле:
   public char symb;
   // Конструктор с двумя аргументами:
   public MyClass(int n, char s){
      code=n;   // Значение целочисленного поля
      symb=s;   // Значение символьного поля
   }
   // Переопределение метода GetHashCode():
   public override int GetHashCode(){
      // Вычисление хэш-кода:
      return code^symb;
   }
   // Переопределение метода Equals():
   public override bool Equals(Object obj){
      // Локальная объектная переменная:
      MyClass t=(MyClass)obj;
      // Результат метода:
      if(code==t.code&&symb==t.symb) return true;
      else return false;
   }
   // Перегрузка оператора "равно":
   public static bool operator==(MyClass a, MyClass b){
      // Вызов метода Equals():
      return a.Equals(b);
```

```
    }
    // Перегрузка оператора "не равно":
    public static bool operator!=(MyClass a, MyClass b){
        // Использование оператора "равно":
        return !(a==b);
    }
}
// Класс с главным методом:
class OverridingEqualsDemo{
    // Главный метод:
    static void Main(){
        // Создание объектов:
        MyClass A=new MyClass(100,'A');
        MyClass B=new MyClass(100,'B');
        MyClass C=new MyClass(200,'A');
        MyClass D=new MyClass(100,'A');
        // Вычисление хэш-кодов:
        Console.WriteLine("Хэш-код A: {0}", A.GetHashCode());
        Console.WriteLine("Хэш-код B: {0}", B.GetHashCode());
        Console.WriteLine("Хэш-код C: {0}", C.GetHashCode());
        Console.WriteLine("Хэш-код D: {0}", D.GetHashCode());
        // Сравнение объектов на предмет
        // равенства и неравенства:
        Console.WriteLine("A==B дает {0}", A==B);
        Console.WriteLine("A!=B дает {0}", A!=B);
        Console.WriteLine("A==C дает {0}", A==C);
        Console.WriteLine("A!=C дает {0}", A!=C);
        Console.WriteLine("A==D дает {0}", A==D);
        Console.WriteLine("A!=D дает {0}", A!=D);
    }
}
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 8.7)

```
Хэш-код A: 37
Хэш-код B: 38
Хэш-код C: 137
Хэш-код D: 37
A==B дает False
A!=B дает True
A==C дает False
A!=C дает True
A==D дает True
A!=D дает False
```

Основу нашего программного кода составляет класс `MyClass`, у которого есть целочисленное поле `code` и символьное поле `symb`. Класс оснащен конструктором с двумя аргументами. Также в классе перегружены операторы == и !=, а также переопределены методы `Equals()` и `GetHashCode()`. Два объекта класса мы будем полагать равными, если у них совпадают значения полей.

Метод `GetHashCode()` в конкретно нашем примере играет декоративную роль (но мы его все равно переопределяем). Относительно остальных участников процесса, то оператор == перегружается так, что вызывается метод `Equals()`, а перегрузка оператора != базируется на использовании оператора ==. Поэтому основа нашего кода — это код метода `Equals()`.

НА ЗАМЕТКУ

В данном случае перегрузка оператора == через вызов метода `Equals()` не есть инструкция к действию. Это просто иллюстрация совместного использования нескольких методов.

Начнем мы с анализа кода `GetHashCode()`. Метод результатом возвращает целое число. Результат вычисляется как значение выражения `code^symb`. Здесь использован оператор «побитового исключающего или» ^, а операндами являются поля `code` и `symb` (для символьного

поля берется код из кодовой таблицы) объекта, из которого вызывается метод. В данном случае не очень важно, какое именно получится число. Важно, чтобы для объектов с одинаковыми значениями полей `code` и `symb` хэш-коды были одинаковыми (при этом случайно могут совпадать хэш-коды разных объектов).

Метод `Equals()` переопределяется так. В теле метода объявляется локальная объектная переменная `t` класса `MyClass`. Значение переменной присваивается командой `t=(MyClass)obj`, где через `obj` обозначен аргумент метода `Equals()`. В этой команде выполняется явное приведение переменной `obj` класса `Object` к ссылке класса `MyClass`. Это означает, что мы собираемся обрабатывать аргумент метода как объектную переменную класса `MyClass`.

ПОДРОБНОСТИ

Мы не определяем «с нуля» метод `Equals()`, а переопределяем его версию, унаследованную из класса `Object`. А там аргумент метода описан как такой, что относится к классу `Object`. При этом мы исходим из того, что при вызове метода аргументом будет передаваться объектная переменная класса `MyClass`. Поэтому в теле метода выполняется явное приведение аргумента к классу `MyClass`. Если при вызове метода передать аргумент другого типа, в общем случае возникает ошибка.

Нередко вместе с переопределением метода `Equals()` выполняется еще и его перегрузка (описывается еще одна версия метода) с аргументом пользовательского класса (в данном случае это класс `MyClass`).

Результат метода вычисляется с помощью условного оператора. Если совпадают значения полей объекта, из которого вызывается метод, и объекта, переданного аргументом методу, то метод `Equals()` результатом возвращает значение `true`, а в противном случае результат метода равен `false`.

Операторный метод для оператора «равно» описывается так, что для аргументов `a` и `b` класса `MyClass` результатом возвращается значение выражения `a.Equals(b)`. А операторный метод для оператора «не равно» возвращает значением выражение `!(a==b)`, что есть «противоположное» значение к результату сравнения объектов `a` и `b` на предмет равенства.

В главном методе программы создается несколько объектов, для этих объектов вычисляется хэш-код, а также объекты сравниваются

на предмет равенства и неравенства. Легко заметить, что объекты с одинаковыми полями интерпретируются как равные.

Перегрузка операторов `true` и `false`

— Как ты посмел обмануть ребенка?!
— Я не могу ждать, пока она вырастет!

из к/ф «Айболит-66»

Операторы `true` и `false` являются унарными и перегружаются в паре (соответствующие операторные методы называются `operator true` и `operator false`). Оба оператора результатом возвращают значение логического типа.

ПОДРОБНОСТИ

Объекты можно проверять на предмет «истинности» или «ложности». При проверке объекта на «истинность» вызывается операторный метод для оператора `true`. Если операторный метод возвращает значение `true`, то такой объект является «истинным». Если операторный метод возвращает значение `false`, то объект не считается «истинным».

При проверке объекта на «ложность» вызывается операторный метод для оператора `false`. Если операторный метод возвращает значение `true`, то объект считается «ложным». Если операторный метод возвращает значение `false`, то объект не считается «ложным».

Строго говоря, если объект не является «истинным», то это не означает, что он «ложный». Из того, что объект не является «ложным», не следует, что объект «истинный». Проверка объекта на «истинность» и «ложность» выполняется в разных ситуациях, которые обсуждаются далее.

Это особые операторы в том смысле, что не сразу понятно, когда и как они применяются. Вместе с тем ситуация не такая уж и сложная. Операторный метод для оператора `true` вызывается в тех случаях, когда объект указан условием — например, в условном операторе или операторе цикла. Также операторный метод для оператора `true` вызывается при проверке первого операнда в выражении на основе оператора `||` (сокращенная форма оператора «логическое или»). Операторный метод для оператора `false` вызывается при проверке первого операнда

в выражении, на основе оператора `&&` (сокращенная форма оператора «логическое и»).

НА ЗАМЕТКУ

Специфику перегрузки операторов `true` и `false` при работе с операторами `||` и `&&` мы обсудим в этой главе, но немного позже.

Небольшой пример, иллюстрирующий перегрузку операторов `true` и `false`, представлен в листинге 8.8.

Листинг 8.8. Перегрузка операторов true и false

```
using System;
// Класс с перегрузкой операторов true и false:
class MyClass{
   // Целочисленное поле:
   public int code;
   // Конструктор с одним аргументом:
   public MyClass(int n){
      // Значение поля:
      code=n;
   }
   // Перегрузка оператора true:
   public static bool operator true(MyClass obj){
      // Проверяется значение поля объекта:
      if(obj.code>=5) return true;
      else return false;
   }
   // Перегрузка оператора false:
   public static bool operator false(MyClass obj){
      // Используется операторный метод для оператора true.
      // Объект использован как условие:
      if(obj) return false;
      else return true;
```

```
   }
}
// Класс с главным методом:
class OverloadingTrueFalseDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта класса:
      MyClass obj=new MyClass(10);
      // Использование операторного метода для
      // оператора true. Объект использован как условие:
      while(obj){
         // Отображение значения поля объекта:
         Console.Write(obj.code+" ");
         // Уменьшение значения поля объекта:
         obj.code--;
      }
      Console.WriteLine();
   }
}
```

Ниже представлен результат выполнения программы.

Результат выполнения программы (из листинга 8.8)

```
10 9 8 7 6 5
```

Мы в классе `MyClass` с целочисленным полем `code` и конструктором (с одним аргументом) описываем еще два операторных метода, которые называются `operator true` и `operator false`. Оба операторных метода описаны с одним аргументом (обозначен как `obj`), являющимся объектом класса `MyClass`. Результатом методами возвращается значение типа `bool`. В методе `operator true()` результат вычисляется так: в условном операторе проверяется условие `obj.code>=5`, состоящее в том, что значение поля `code` объекта-операнда больше или равно `5`. Если так, то результатом метода возвращается значение `true`. В противном случае метод возвращает значение `false`.

НА ЗАМЕТКУ

Получается так, что если у объекта значение целочисленного поля больше или равно 5, то при проверке на «истинность» этот объект признается «истинным». Если значение поля меньше 5, то при проверке на истинность объект «истинным» не признается.

Описание метода `operator false()` тоже базируется на использовании условного оператора. Но на этот раз условием в условном операторе указан объект `obj` (аргумент метода). Если объект указан в качестве условия, то проверка условия сводится к проверке объекта на «истинность»: если объект признается «истинным», то это интерпретируется как истинность условия, а если объект истинным не признается, то условие будет интерпретироваться как ложное. Проще говоря, объект `obj` в условии в условном операторе означает, что для этого объекта будет вызван операторный метод `operator true()`. Результат, возвращаемый методом, дает значение условия в условном операторе. Если объект `obj` истинный, то результатом метода `operator false()` возвращается значение `false`. Если объект не является истинным, то результатом метода `operator false()` возвращается значение `true`.

НА ЗАМЕТКУ

Мы определили методы `operator true()` и `operator false()` так, что если один метод возвращает значение `true`, то другой возвращает значение `false`, и наоборот. Получается, что если объект не «истинный», то он «ложный», а если объект не «ложный», то он «истинный». Но в общем случае все могло бы быть иначе.

В главном методе программы создается объект `obj` класса `MyClass`, и целочисленное поле этого объекта получает значение `10`. После этого выполняется оператор цикла `while`, в котором условием указан объект `obj`. Каждый раз при проверке условия для объекта `obj` вызывается операторный метод `operator true()`. Пока значение поля объекта больше или равно `5`, объект интерпретируется как «истинный» и условие в операторе цикла тоже истинно. За каждый цикл командой `Console.Write(obj.code+" ")` текущее значение поля `code` объекта `obj` отображается в консольном окне, после чего командой `obj.code--` значение поля уменьшается на единицу. В результате в консольном окне появляется последовательность числовых значений от `10` до `5` включительно.

Перегрузка логических операторов

Я же говорил, что я всегда бываю нрав!

из к/ф «Айболит-66»

Бинарные операторы `&` и `|` и унарный оператор `!` могут перегружаться произвольным образом. Причем тип результата для соответствующих операторных методов не обязательно должен быть логическим значением. Операторы `&&` и `||` не перегружаются (то есть их нельзя перегрузить). Но в классе можно так перегрузить операторы `&` и `|`, а также операторы `true` и `false`, что с объектами класса станет возможно использовать операторы `&&` и `||`. Именно такой способ перегрузки операторов `&`, `|`, `true` и `false` мы и рассмотрим.

Чтобы уловить идею, на которой основан весь подход по перегрузке операторов, имеет смысл проанализировать способ вычисления выражений вида `A&&B` и `A||B`. Мы исходим из того, что операнды `A` и `B` являются объектами некоторого класса (в котором мы планируем перегружать операторы).

Итак, при вычислении выражения `A&&B` первый операнд `A` проверяется на предмет ложности. Для этого вызывается операторный метод для оператора `false`. Если операнд `A` «ложный» (операторный метод `operator false()` возвращает значение `true`), то результатом выражения `A&&B` возвращается операнд `A`. Если операнд `A` «ложным» не признается (операторный метод `operator false()` возвращает значение `false`), то результатом выражения `A&&B` возвращается значение выражения `A&B`.

При вычислении значения выражения `A||B` операнд `A` проверяется на предмет «истинности», и для этого вызывается операторный метод `operator true()`. Если метод возвращает значение `true`, то результатом выражения `A||B` возвращается операнд `A`. Если операторный метод `operator true()` возвращает значение `false`, то результатом выражения `A||B` возвращается результат выражения `A|B`.

Отсюда становятся понятны принципы, определяющие способ перегрузки операторов `&`, `|`, `true` и `false` для возможности использования операторов `&&` и `||`. Они таковы:

- Операторы `&` и `|` должны быть перегружены для пользовательского класса, причем операнды должны быть объектами этого класса, и результат — объект этого же класса.

- Для пользовательского класса необходимо описать операторные методы для операторов `true` и `false`.

Как эта схема реализуется на практике, показано в программе в листинге 8.9.

Листинг 8.9. Перегрузка логических операторов

```
using System;
// Класс с перегрузкой операторов:
class MyClass{
   // Символьное поле:
   public char symb;
   // Конструктор с одним аргументом:
   public MyClass(char s){
      symb=s;   // Значение поля
   }
   // Перегрузка оператора true:
   public static bool operator true(MyClass obj){
      switch(obj.symb){
         case 'A':
         case 'E':
         case 'I':
         case 'O':
         case 'U':
         case 'Y':
            return true;
         default:
            return false;
      }
   }
   // Перегрузка оператора false:
   public static bool operator false(MyClass obj){
      if(obj) return obj.symb=='Y';
```

```
      else return true;
   }
   // Перегрузка оператора &:
   public static MyClass operator&(MyClass a, MyClass b){
      if(a) return b;
      else return a;
   }
   // Перегрузка оператора |:
   public static MyClass operator|(MyClass a, MyClass b){
      if(a) return a;
      else return b;
   }
}
// Класс с главным методом:
class OverloadingLogOpsDemo{
   // Главный метод:
   static void Main(){
      // Создание объектов:
      MyClass A=new MyClass('A');
      MyClass B=new MyClass('B');
      MyClass E=new MyClass('E');
      MyClass Y=new MyClass('Y');
      // Использование логических операторов:
      Console.WriteLine("Выражение A&&B: {0}",(A&&B).symb);
      Console.WriteLine("Выражение B&&A: {0}",(B&&A).symb);
      Console.WriteLine("Выражение A&&E: {0}",(A&&E).symb);
      Console.WriteLine("Выражение E&&A: {0}",(E&&A).symb);
      Console.WriteLine("Выражение A&&Y: {0}",(A&&Y).symb);
      Console.WriteLine("Выражение Y&&A: {0}",(Y&&A).symb);
      Console.WriteLine("Выражение A||B: {0}",(A||B).symb);
      Console.WriteLine("Выражение B||A: {0}",(B||A).symb);
      Console.WriteLine("Выражение A||E: {0}",(A||E).symb);
```

```
        Console.WriteLine("Выражение E||A: {0}",(E||A).symb);
        Console.WriteLine("Выражение A||Y: {0}",(A||Y).symb);
        Console.WriteLine("Выражение Y||A: {0}",(Y||A).symb);
    }
}
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 8.9)

```
Выражение A&&B: B
Выражение B&&A: B
Выражение A&&E: E
Выражение E&&A: A
Выражение A&&Y: Y
Выражение Y&&A: Y
Выражение A||B: A
Выражение B||A: A
Выражение A||E: A
Выражение E||A: E
Выражение A||Y: A
Выражение Y||A: Y
```

Класс `MyClass` имеет одно поле (символьное поле с названием `symb`) и конструктор с одним аргументом (определяет значение поля). В классе описаны четыре операторных метода. В теле операторного метода `operator true()` с помощью оператора выбора проверяется значение символьного поля объекта-операнда. Если это большая английская гласная буква, то метод возвращает результатом `true`. В противном случае результат равен `false`.

В теле операторного метода `operator false()` есть условный оператор. В условном операторе условием указан объект `obj` (аргумент метода). Если объект «истинный» (а это определяется вызовом операторного метода `operator true()`), то результатом метода `operator false()` возвращается значение выражения `obj.symb=='Y'` (равно `false` за исключением случая, когда значение поля равно `'Y'`). Если объект `obj` не является «истинным», то результатом метода `operator false()` возвращается значение `true`.

НА ЗАМЕТКУ

Получается, что если значением поля объекта является один из символов 'A', 'E', 'I', 'O', 'U' или 'Y', то объект интерпретируется как «истинный». Как «ложный» интерпретируется любой не являющийся «истинным» объект, а также объект со значением 'Y' символьного поля. Другими словами, если у объекта поле равно 'Y', то такой объект интерпретируется и как «истинный», и как «ложный». Такой вот парадокс (законный тем не менее).

Операторный метод `operator&()` описан так, что если первый аргумент (объект `a` класса `MyClass`) «истинный», то результатом возвращается ссылка на второй аргумент (объект `b` класса `MyClass`). Если первый операнд не «истинный», то результатом возвращается ссылка на него.

Операторный метод `operator|()` возвращает ссылку на первый операнд, если он «истинный». Если первый операнд не «истинный», то результатом метода возвращается ссылка на второй операнд.

В главном методе программы создается четыре объекта (название объекта совпадает с символом, являющимся значением символьного поля объекта). Объекты используются для вычисления выражений на основе операторов `&&` и `||`. Для проверки выводится значение поля `symb` объекта-результата.

ПОДРОБНОСТИ

Кратко проанализируем результат выполнения операций с операторами `&&` и `||`.

- При вычислении выражения `A&&B` объект `A` проверяется на «ложность». Объект не «ложный», поэтому результатом возвращается значение выражения `A&B`. При вычислении значения выражения `A&B` объект `A` проверяется на «истинность». Он «истинный», поэтому результатом возвращается ссылка на объект `B`.
- При вычислении выражения `B&&A` на «ложность» проверяется объект `B`. Он «ложный», поэтому результатом возвращается ссылка на объект `B`.
- При вычислении выражения `A&&E` на «ложность» проверяется первый операнд `A`. Он не «ложный», поэтому результатом возвращается значение выражения `A&E`. При вычислении выражения `A&E` объект `A` проверяется на «истинность». Объект «истинный», и поэтому результатом выражения `A&E` является ссылка на объект `E`.

- При вычислении выражения `E&&A` на «ложность» проверяется объект `E`. Он не «ложный», и поэтому результатом выражения возвращается выражение `E&A`. В данном выражении на «истинность» проверяется объект `E`. Он «истинный», поэтому результатом возвращается ссылка на объект `A`.
- При вычислении выражения `A&&Y` проверяется на «ложность» объект `A`, и поскольку он не «ложный», то результатом является значение выражения `A&Y`. Поскольку объект `A` «истинный», то значение выражения `A&Y` — это ссылка на объект `Y`.
- При вычислении выражения `Y&&A` на «ложность» проверяется объект `Y`. Он «ложный», поэтому результатом является ссылка на объект `Y`.
- При вычислении выражения `A||B` объект `A` проверяется на «истинность». Он «истинный», поэтому результатом является ссылка на `A`.
- При вычислении выражения `B||A` объект `B` проверяется на «истинность». Объект не «истинный», поэтому результат вычисляется как значение выражения `B|A`. Поскольку объект `B` не «истинный», то результат последнего выражения — ссылка на объект `A`.
- Вычисление выражения `A||E` начинается с проверки на «истинность» объекта `A`. Он «истинный», поэтому ссылка на объект `A` возвращается как значение выражения.
- При вычислении выражения `E||A` на «истинность» проверяется объект `E`. Поскольку объект `E` «истинный», ссылка на него возвращается как результат.
- Выражение `A||Y` вычисляется проверкой на «истинность» объекта `A`. Объект «истинный». Поэтому результатом является ссылка на объект `A`.
- При вычислении значения выражения `Y||A` на «истинность» проверяется объект `Y`. Объект `Y` «истинный», и ссылка на него является значением выражения.

Перегрузка операторов приведения типов

> Пустите доброго человека! Пустите доброго человека, а не то он выломает дверь!
>
> *из к/ф «Айболит-66»*

Приведение (преобразование) типов состоит в том, что значение одного типа преобразуется в значение другого типа. Есть два типа преобразования: *явное* и *неявное*. При явном преобразовании типа перед значением (преобразуемым к другому типу) в круглых скобках указывается название типа, к которому выполняется преобразование. Неявное

преобразование выполняется автоматически в случаях, когда по контексту некоторой команды в определенном месте должно быть значение определенного типа, а по факту указано значение другого типа.

НА ЗАМЕТКУ

Конечно, мы исходим из того, что преобразование возможно в принципе.

Случаи, когда выполняется неявное преобразование, и правила выполнения явного и неявного преобразований для базовых типов предопределены. Если необходимо выполнять преобразования, в которых исходный тип (значение этого типа преобразуется) или конечный тип (тип, к которому преобразуется значение) относятся к пользовательскому классу, мы можем определить способ такого преобразования, описав в классе соответствующий операторный метод.

Операторные методы, определяющие способ преобразования из пользовательского типа или в пользовательский тип, описываются следующим образом и с учетом таких правил.

- Операторный метод для выполнения приведения (преобразования) типов описывается как унарный. Аргумент операторного метода отождествляется с преобразуемым значением.
- «Название» операторного метода состоит из двух «слов»: ключевого слова `operator` и идентификатора типа, к которому выполняется преобразование. При этом тип результата для операторного метода не указывается (он совпадает с идентификатором типа после ключевого слова `operator`).
- Один из двух типов (тип аргумента или тип, к которому выполняется преобразование) должен быть классом, в котором описан операторный метод.
- При описании операторного метода для выполнения приведения типа используется ключевое слово `implicit` или `explicit`. Если оператор описан с ключевым словом `implicit`, то это оператор для неявного приведения типа. Если оператор описан с ключевым словом `explicit`, то такой оператор определяет явное приведение типа.
- Может быть описан операторный метод только для одной формы приведения типа (явной или неявной). Если определено неявное

приведение типа, то в таком случае явное приведение также определено (выполняется тем же операторным методом, что и неявное приведение).

Пример, в котором иллюстрируется работа операторных методов для выполнения приведения типов, представлен в листинге 8.10.

Листинг 8.10. Перегрузка операторов приведения типа

```
using System;
// Класс с перегрузкой операторов приведения типа:
class MyClass{
   public int code;     // Целочисленное поле
   public char symb;    // Символьное поле
   public String text;  // Текстовое поле
   // Конструктор с тремя аргументами:
   public MyClass(int n, char s, String t){
      code=n;   // Значение числового поля
      symb=s;   // Значение символьного поля
      text=t;   // Значение текстового поля
   }
   // Переопределение метода ToString():
   public override String ToString(){
      // Локальная текстовая строка:
      String txt="Поля объекта:\n";
      txt+="Числовое поле: "+code+"\n";
      txt+="Символьное поле: \'"+symb+"\'\n";
      txt+="Текстовое поле: \""+text+"\"\n";
      txt+="-----------------------";
      // Результат метода:
      return txt;
   }
   // Метод для явного приведения к текстовому типу:
   public static explicit operator String(MyClass obj){
      return obj.text;
```

```
   }
   // Метод для неявного приведения к типу int:
   public static implicit operator int(MyClass obj){
      return obj.code;
   }
   // Метод для неявного приведения к типу char:
   public static implicit operator char(MyClass obj){
      return obj.symb;
   }
   // Метод для неявного преобразования из типа int:
   public static implicit operator MyClass(int n){
      MyClass t=new MyClass(n,'B',"Bravo");
      return t;
   }
   // Метод для явного преобразования из типа char:
   public static explicit operator MyClass(char s){
      return new MyClass(300, s,"Charlie");
   }
   // Метод для неявного преобразования из текстового типа:
   public static implicit operator MyClass(String t){
      return new MyClass(t.Length, t[0], t);
   }
}
// Класс с главным методом:
class OverloadingImplExplDemo{
   // Главный метод:
   static void Main(){
      // Создание объектов и проверка результата.
      // Явное создание объекта:
      MyClass A=new MyClass(100,'A',"Alpha");
      // Неявно вызывается метод ToString():
      Console.WriteLine("Объект A. "+A);
```

```
        // Создание объекта преобразованием из типа int:
        MyClass B=200;
        // Неявно вызывается метод ToString():
        Console.WriteLine("Объект B. "+B);
        // Создание объекта преобразованием из типа char:
        MyClass C=(MyClass)'C';
        // Неявно вызывается метод ToString():
        Console.WriteLine("Объект C. "+C);
        // Создание объекта преобразованием из текста:
        MyClass D="Delta";
        // Неявно вызывается метод ToString():
        Console.WriteLine("Объект D. "+D);
        Console.WriteLine("Еще раз поля объекта A:");
        // Явное преобразование в тип int:
        Console.WriteLine("Число: "+(int)A);
        // Явное преобразование в тип char:
        Console.WriteLine("Символ: "+(char)A);
        // Явное преобразование в текст:
        Console.WriteLine("Текст: "+(String)A+"\n");
        Console.WriteLine("Разные операции:");
        // Целочисленная переменная:
        int n;
        // Неявное преобразование к типу int:
        n=A+B;
        Console.WriteLine("Значение A+B="+n);
        // Неявное преобразование к типу char:
        char s=B;
        Console.WriteLine("Символ: "+s);
        // Последовательное преобразование из текстового
        // типа к типу MyClass, а затем к типу int:
        Console.WriteLine("Число: "+(int)(MyClass)"Echo");
    }
}
```

При выполнении программы получаем такой результат.

Результат выполнения программы (из листинга 8.10)

```
Объект A. Поля объекта:
Числовое поле: 100
Символьное поле: 'A'
Текстовое поле: "Alpha"
-----------------------
Объект B. Поля объекта:
Числовое поле: 200
Символьное поле: 'B'
Текстовое поле: "Bravo"
-----------------------
Объект C. Поля объекта:
Числовое поле: 300
Символьное поле: 'C'
Текстовое поле: "Charlie"
-----------------------
Объект D. Поля объекта:
Числовое поле: 5
Символьное поле: 'D'
Текстовое поле: "Delta"
-----------------------
Еще раз поля объекта A:
Число: 100
Символ: A
Текст: Alpha

Разные операции:
Значение A+B=300
Символ: B
Число: 4
```

Проанализируем программный код и результат его выполнения. Мы имеем дело с классом `MyClass` с тремя полями (целочисленным, символьным и текстовым). У класса есть конструктор с тремя аргументами, в классе переопределен метод `ToString()`, а еще там описано несколько операторных методов для выполнения приведения типов. Мы определяем такие способы приведения типов:

- При явном преобразовании объекта класса `MyClass` в значение типа `String` (инструкция `explicit operator String(MyClass obj)` в описании метода) результатом является текстовое поле `text` объекта.
- Неявное преобразование объекта класса `MyClass` в значение типа `int` (инструкция `implicit operator int(MyClass obj)` в описании метода) состоит в том, что результатом возвращается значение целочисленного поля `code` объекта.
- При неявном преобразовании объекта класса `MyClass` к типу `char` (инструкция `implicit operator char(MyClass obj)` в описании метода) результатом возвращается значение символьного поля `symb` объекта.
- В программе описан метод для неявного преобразования значения типа `int` в объект класса `MyClass` (инструкция `implicit operator MyClass(int n)` в описании метода). В теле операторного метода командой `MyClass t=new MyClass(n,'B',"Bravo")` создается объект класса `MyClass`, после чего этот объект возвращается результатом (это и есть результат приведения типов). Таким образом, число, преобразуемое к объекту класса `MyClass`, определяет значение поля `code` этого объекта. Символьное поле у объекта будет равно `'B'`, а текстовое поле объекта получает значение `"Bravo"`.
- Метод для явного преобразования значения типа `char` в объект класса `MyClass` (инструкция `explicit operator MyClass(char s)` в описании метода) возвращает результатом объект, который создается командой `new MyClass(300, s,"Charlie")`. Это означает, что преобразуемое к объекту класса `MyClass` символьное значение задает значение символьного поля `symb` созданного объекта, целочисленное поле `code` этого объекта равно `300`, а текстовое поле `text` объекта есть `"Charlie"`.
- Метод для неявного преобразования текстового значения в объект класса `MyClass` (инструкция `implicit operator MyClass(String t)` в описании метода) результатом возвращает

объект, создаваемый командой `new MyClass(t.Length, t[0], t)`. Целочисленное поле этого объекта — количество символов в текстовом значении, преобразуемом к объекту класса `MyClass`. Символьное поле объекта — это первый символ в тексте. Текстовое поле объекта — собственно текст, преобразуемый к объекту класса `MyClass`.

В главном методе программы проверяется работа операторных методов для выполнения явного и неявного приведения типов. Объект `A` создается обычным способом, с вызовом конструктора. Еще три объекта (`B`, `C` и `D`) создаются с использованием операторов приведения типа. Так, при выполнении команды `MyClass B=200`, которой формально объектной переменной класса `MyClass` присваивается целое число, вызывается операторный метод для приведения целочисленного типа к типу `MyClass`. В результате создается объект, и ссылка на него записывается в переменную `B`. Похожая ситуация имеет место при выполнении команды `MyClass C=(MyClass)'C'`, но поскольку операторный метод описан для явного приведения типа `char` к типу `MyClass`, то в данном случае приведение выполняется явно — перед символьным значением `'C'`, которое присваивается значением объектной переменной, указана инструкция `(MyClass)` явного приведения типа.

ПОДРОБНОСТИ

Если в команде `MyClass C=(MyClass)'C'` не использовать инструкцию явного приведения типа (то есть если воспользоваться командой `MyClass C='C'`), то произойдет следующее. Неявного преобразования из типа `char` в тип `MyClass` нет. Зато есть автоматическое (встроенное) приведение типа `char` к типу `int`. А в классе `MyClass` описан оператор неявного приведения типа `int` к типу `MyClass`. Поэтому при выполнении команды `MyClass C='C'` символьное значение `'C'` будет сначала преобразовано к типу `int` (значение `67`), а затем будет задействован оператор неявного приведения типа `int` к типу `MyClass`.

Еще один объект создан командой `MyClass D="Delta"`. Здесь объектной переменной значением присваивается текст, поэтому вызывается оператор неявного приведения значения класса `String` к значению класса `MyClass`: создается объект, и ссылка на него записывается в переменную `D`.

В классе `MyClass` описан метод `ToString()` и оператор явного приведения к текстовому типу. Результат у них разный. При выполнении

команд вида "Объект A. "+A, когда к тексту прибавляется объект класса MyClass, для преобразования объекта в текст вызывается оператор ToString(). Примером явного приведения объекта класса MyClass к текстовому формату может быть инструкция (String)A. Здесь инструкция приведения к типу String указана явно перед именем объекта.

НА ЗАМЕТКУ

У созданного объекта значение поля code равно 5 (количество символов в тексте "Delta"), значение поля symb равно 'D' (первый символ в тексте "Delta"), а значение поля text равно "Delta".

Инструкции (int)A и (char)A являются примером явного преобразования объекта класса MyClass в значение типа int и значение типа char соответственно. А вот когда выполняется команда n=A+B, которой целочисленной переменной n значением присваивается сумма двух объектов класса MyClass, выполняется автоматическое (неявное) преобразование объектов к целочисленному типу. В итоге значение переменной n — это сумма целочисленных полей объектов A и B. Неявное преобразование к типу char имеет место и при присваивании символьной переменной s значением объекта B (команда char s=B).

ПОДРОБНОСТИ

Если мы описали оператор явного приведения типа, то приведение типа может выполняться только в явном виде. Если описан оператор неявного приведения типа, то приведение типа может выполняться и в явном, и в неявном виде. Хотя в программе описан оператор явного приведения к текстовому типу, неявное преобразование также выполняется. Но в этом случае вызывается переопределенный в классе MyClass метод ToString().

Еще одна инструкция, достойная внимания, имеет вид (int)(MyClass) "Echo". Здесь имеют место два последовательных явных приведения типа. Сначала текст приводится к типу MyClass (инструкция (MyClass)"Echo"). Результатом является объект класса MyClass. Его целочисленное поле code имеет значение 4 (количество символов в тексте "Echo"), значение поля symb равно 'E' (первый символ в тексте "Echo"), а текстовое поле text имеет значение "Echo". При явном приведении этого объекта к целочисленному типу получаем значение 4 (значение поля code объекта).

Команды присваивания и перегрузка операторов

— Но ведь ты же обещал!
— Ну мало ли что я наобещаю! Знаешь, какой я подлый? Я обещаю одно, а делаю совершенно другое!

из к/ф «Айболит-66»

В начале главы утверждалось, что сокращенные формы оператора присваивания (+=, *=, /= и так далее) не перегружаются, но можно так перегрузить базовые операторы (например, +, * или /), что сокращенные формы операторов можно будет использовать в программе. Главный критерий здесь состоит в том, чтобы тип результата, возвращаемого операторным методом для базового оператора, соответствовал типу переменной, которой присваивается значение с помощью сокращенной формы оператора присваивания. Но и здесь не все так очевидно. Как пример, иллюстрирующий сказанное, рассмотрим программу в листинге 8.11.

Листинг 8.11. Операции присваивания и перегрузка операторов

```
using System;
// Класс с перегрузкой операторов:
class MyClass{
   // Целочисленное поле:
   public int code;
   // Конструктор с одним аргументом:
   public MyClass(int n){
      code=n;   // Присваивание значения полю
   }
   // Операторный метод для сложения объектов:
   public static MyClass operator+(MyClass a, MyClass b){
      return new MyClass(a.code+b.code);
   }
   // Операторный метод для умножения объектов:
   public static int operator*(MyClass a, MyClass b){
```

```
      return a.code*b.code;
   }
   // Операторный метод для неявного приведения типа
   // (из типа int к типу MyClass):
   public static implicit operator MyClass(int n){
      return new MyClass(n);
   }
}
// Класс с главным методом:
class OverloadingAndAssigningDemo{
   static void Main(){
      // Создание объекта:
      MyClass A=new MyClass(7);
      // Проверка результата:
      Console.WriteLine("Объект A: {0}", A.code);
      // Создание объекта:
      MyClass B=new MyClass(8);
      // Проверка результата:
      Console.WriteLine("Объект B: {0}", B.code);
      // Вычисление суммы объектов с использованием
      // сокращенной формы оператора присваивания:
      A+=B;
      // Проверка результата:
      Console.WriteLine("Объект A: {0}", A.code);
      // Вычисление произведения объектов с использованием
      // сокращенной формы оператора присваивания:
      A*=B;
      // Проверка результата:
      Console.WriteLine("Объект A: {0}", A.code);
   }
}
```

Результат выполнения программы такой, как показано ниже.

Результат выполнения программы (из листинга 8.11)

```
Объект A: 7
Объект B: 8
Объект A: 15
Объект A: 120
```

В классе `MyClass` (с числовым полем и конструктором с одним аргументом) описаны операторные методы для вычисления суммы объектов класса `MyClass`, произведения объектов класса `MyClass`, а также оператор для неявного приведения целочисленного значения к объекту класса `MyClass`. В главном методе программы создаются два объекта `A` и `B` класса `MyClass`, после чего выполняются команды `A+=B` и `A*=B`.

При выполнении команды `A+=B` сначала вычисляется сумма `A+B`. Результатом является объект, целочисленное поле которого равно сумме полей объектов `A` и `B`. Ссылка на этот объект записывается в переменную `A`.

При выполнении команды `A*=B` сначала вычисляется произведение `A*B`. Значением этого выражения является целое число, равное произведению значений полей объектов `A` и `B`. Вообще, число присвоить значением объектной переменной нельзя. Но в нашем случае в классе `MyClass` описан оператор неявного приведения типа `int` к типу `MyClass`. Этот операторный метод будет задействован, и на основе числового значения `A*B` создается объект, а ссылка на этот объект записывается в переменную `A`.

Резюме

> Солнце есть. Песок есть. Притяжение есть. Где мы? Мы на Земле.
>
> *из к/ф «Кин-дза-дза»*

- В языке C# можно определять действие операторов на операнды, являющиеся объектами пользовательских классов. Этот механизм называется перегрузкой операторов и реализуется путем описания в пользовательском классе операторных методов.
- Операторный метод описывается с ключевыми словами `public` и `static` — то есть метод должен быть открытым и статическим. Операторный метод обязательно должен возвращать результат.

Название операторного метода получается объединением ключевого слова `operator` и символа оператора. Аргументы операторного метода отождествляются с операндами выражения на основе перегружаемого оператора. У операторных методов для бинарных операторов должно быть два аргумента, у операторных методов для унарных операторов — один аргумент. Хотя бы один аргумент операторного метода должен быть объектом класса, в котором этот операторный метод описан.

- Операторы сравнения перегружаются парами (< и >, <= и >=, == и !=). При перегрузке операторов == и != обычно переопределяют и методы `Equals()` и `GetHashCode()`.
- Операторы `true` и `false` используются для проверки объектов на «истинность» и «ложность». Операторы перегружаются в паре и должны возвращать результатом логическое значение. Оператор `true` вызывается, если объект указан условием в условном операторе или операторе цикла, а также при проверке первого операнда в выражении на основе оператора `||`. Оператор `false` вызывается при проверке первого операнда в выражении на основе оператора `&&`.
- Операторы `&&` и `||` не перегружаются. Но есть способ так перегрузить операторы `&`, `|`, `true` и `false`, что операторы `&&` и `||` можно будет использовать.
- Можно описать операторные методы для выполнения явного и неявного приведения типов (при условии, что одним из типов является пользовательский класс). Операторы для явного приведения типов описываются с ключевым словом `explicit`, операторы для неявного приведения типов описываются с ключевым словом `implicit`.
- Сокращенные формы операторов присваивания не перегружаются. Но можно так перегрузить базовые операторы, что сокращенные формы операторов также будут рабочими.

Задания для самостоятельной работы

— Мы договорились?
— Да, нринц!
— Значит, я ставлю ультиматум.
— Да. А я захожу сзади.

из к/ф «Формула любви»

1. Напишите программу, в которой есть класс с символьным полем и следующими перегруженными операторами: оператором инкремента ++ и декремента --, бинарным оператором «плюс» + и бинарным оператором «минус» -. Правила перегрузки операторов такие. Применение оператора инкремента к объекту приводит к тому, что поле значением получает следующий (по отношению к текущему значению) символ в кодовой таблице. Применение оператора декремента приводит к тому, что поле получает значением предыдущий (по отношению к текущему значению) символ в кодовой таблице. Значением выражения на основе оператора инкремента/декремента является ссылка на объект-операнд. Бинарный оператор «плюс» можно применять для вычисления суммы объекта и целого числа, а также суммы целого числа и объекта. В обоих случаях результатом возвращается новый объект, значение символьного поля которого определяется прибавлением целого числа (один из операндов) к коду символа из объекта-операнда. С помощью бинарного оператора «минус» можно вычислять разность двух объектов. Результатом является целое число — разность кодов символов из объектов-операндов.

2. Напишите программу, в которой есть класс с полем, являющимся ссылкой на одномерный целочисленный массив. У класса есть конструктор с одним целочисленным аргументом, определяющим размер массива. При создании объекта все элементы массива получают нулевые значения. В классе перегружаются следующие операторы. Унарный оператор ~ перегружен таким образом, что результатом возвращается текстовая строка со значениями элементов массива (на который ссылается поле объекта, к которому применяется оператор). Унарный оператор инкремента ++ перегружен так, что его применение к объекту приводит к добавлению в массив нового элемента с нулевым значением. Результатом возвращается ссылка на объект-операнд. При применении к объекту оператора декремента -- из массива удаляется один элемент (например, последний), а результатом возвращается ссылка на объект-операнд. Бинарный оператор сложения + должен быть определен так, чтобы можно

было вычислять сумму двух объектов, объекта и числа, а также числа и объекта. Во всех случаях результатом возвращается новый объект. Если в операции участвуют два объекта-операнда, то в объекте-результате массив формируется объединением массивов складываемых объектов. Если вычисляется сумма объекта и числа, то в объекте-результате массив получается добавлением нового элемента к массиву из объекта-операнда. Значение добавляемого элемента определяется значением числа-операнда. Если к числу прибавляется объект, то новый элемент добавляется в начало массива. Если к объекту прибавляется число, то новый элемент добавляется в конец массива.

3. Напишите программу, в которой есть класс с двумя целочисленными полями. Опишите для этого класса операторные методы, которые позволяют сравнивать объекты класса на предмет «меньше» или «больше». Исходите из того, что один объект меньше/больше другого, если сумма квадратов значений его полей меньше/больше суммы квадратов значений полей другого объекта.

4. Напишите программу, в которой есть класс с целочисленным полем и текстовым полем. Выполните перегрузку всех операторов сравнения. Сравнение на предмет «больше» или «меньше» выполняется на основе сравнения длины текстовых значений (имеются в виду текстовые поля сравниваемых объектов). При сравнении на предмет «больше или равно» или «меньше или равно» сравниваются значения целочисленных полей объектов. При сравнении на предмет «равно» или «не равно» сравниваются и целочисленные, и текстовые поля объектов. Также предложите способ переопределения методов `Equals()` и `GetHashCode()`.

5. Напишите программу, в которой есть класс с целочисленным полем и символьным полем. Перегрузите операторы `true` и `false` так, чтобы «истинным» считался объект, у которого разность значения целочисленного поля и кода символа из символьного поля не превышает величину `10`. Используйте объект данного класса (в качестве условия в условном операторе) для того, чтобы отобразить последовательность символов в консольном окне.

6. Напишите программу, в которой есть класс с целочисленным полем. Перегрузите операторы `&`, `|`, `true` и `false` так, чтобы с объектами класса можно было использовать операторы `&&` и `||`. Перегрузку следует реализовать так, чтобы объект считался «истинным», если значение его числового поля равно `2`, `3`, `5` или `7`. Объект должен рассматриваться как «ложный», если значение его числового поля меньше `1` или больше `10`.

7. Напишите программу, в которой есть класс с текстовым полем. Опишите в классе операторные методы для выполнения приведения типов. Необходимо определить следующие способы преобразований. При преобразовании объекта в целое число результатом возвращается количество символов в значении текстового поля. При преобразовании объекта в символ результатом является первый символ в тексте. При преобразовании числа в объект создается (и возвращается результатом соответствующего операторного метода) объект, текстовое поле которого содержит текстовую строку из символов `'A'`. Количество символов в тексте определяется преобразуемым числом.

8. Напишите программу, в которой есть класс с полем, являющимся ссылкой на целочисленный массив. Опишите в классе операторные методы для выполнения приведений типов. Необходимо реализовать следующие правила приведения типов. При преобразовании объекта в текст возвращается текстовая строка со значениями элементов массива. При преобразовании объекта в число возвращается сумма элементов массива. При преобразовании числа в объект результатом является новый объект, размер массива в котором определяется преобразуемым числом. Массив в объекте должен быть заполнен нулями.

9. Напишите программу, в которой есть класс с целочисленным полем и перегрузкой операторов +, – и *. Предложите такой способ перегрузки этих операторов, чтобы с объектами класса можно было использовать операторы +=, –= и *=.

10. Напишите программу, в которой есть класс с символьным полем и перегрузкой операторов + и –. Операторы должны быть перегружены так, чтобы применение оператора + к объектам класса давало результатом текст, получающийся объединением значений символьных полей суммируемых объектов. При применении оператора – к объектам класса результатом должно возвращаться целое число (разность кодов символов из вычитаемых объектов). Предложите такие способы перегрузки операторов приведения типа, чтобы с объектами класса можно было использовать операторы += и –=.

Глава 9
СВОЙСТВА И ИНДЕКСАТОРЫ

— Друг, у вас какая система? Разрешите взглянуть?
— Система обычная. Нажал на кнопку — и дома.

из к/ф «Кин-дза-дза»

Эта глава посвящена двум достаточно экзотическим членам классов, которые встречаются (и очень часто используются) в языке C#. Речь пойдет о *свойствах* и *индексаторах*. В частности, нам предстоит узнать:

- что такое свойство и как оно описывается;
- как определить режимы доступа к свойству;
- что такое индексатор и как он описывается;
- как задается режим использования индексатора;
- как создается многомерный индексатор.

Также мы рассмотрим немалое количество примеров, объясняющих и иллюстрирующих пользу, красоту и эффективность свойств и индексаторов.

Знакомство со свойствами

Пусть они посмотрят, сколько у меня всякого добра! О, у меня еще в чулане сколько!

из к/ф «Айболит-66»

Мы начинаем знакомство со *свойствами*. Свойство — это член класса, как поле или метод. Но это и не поле, и не метод. Свойство в языке C# — это нечто среднее между полем и методом. Если исходить из того, как свойство используется, то оно, конечно, очень напоминает

поле. Но принципы реализации свойства больше напоминают описание методов. Чтобы было легче понять, как свойство описывается в классе, начнем с того, как оно используется.

Используется свойство, как отмечалось выше, практически так же, как поле. Для большей конкретики представим, что существует некий объект и у этого объекта есть поле (например, числовое). Что мы можем делать с этим полем? В принципе, есть две базовые операции: считывание значения поля и присваивание значения полю. И в том, и в другом случае мы указываем имя объекта и название поля (через точку). Реакция на такую инструкцию достаточно простая: выполняется обращение к области памяти, в которой хранится значение поля. Если в команде значение поля считывается, то это значение будет прочитано в соответствующей области памяти. Если значение полю присваивается, то значение будет записано в соответствующую область памяти. Подчеркнем, что все это для поля. В случае со свойством формально все выглядит очень похоже. Как и у поля, у свойства есть тип и есть название. Если мы хотим узнать значение свойства, то действуем подобно случаю с полем: после имени объекта через точку указывается название свойства. Если мы хотим присвоить значение свойству, то указываем имя объекта, название свойства (через точку) и, после оператора присваивания, присваиваемое значение. То есть все, как и в случае с полем. В чем же разница? А разница в действиях, которые выполняются при считывании значения свойства и при присваивании значения свойству.

При описании свойства определяются два специальных метода, которые непосредственно связаны со свойством. Эти методы называются *аксессорами*. Один метод (называется *get-аксессором*) вызывается при считывании значения свойства. Другой метод (называется *set-аксессором*) вызывается при присваивании значения свойству. Проще говоря, при считывании значения свойства и при присваивании значения свойству вызываются специальные методы (в отличие от случая с полем, когда обращение к полю означает обращение к области памяти, в которой хранится значение поля). Что будет происходить при вызове методов-аксессоров, определяем мы, когда описываем свойство в классе. Это может быть банальное считывание/присваивание значения некоторому полю, определенный вычислительный процесс, обращение к массиву или другому объекту — все зависит от нашей фантазии и той задачи, которая решается. В этом смысле свойство близко к методу, причем не одному методу, а сразу двум методам. Считывание значения свойства означает вызов метода, возвращающего значение (значение свойства). Присваивание значения

свойству подразумевает вызов другого метода, который может присваивать значения полям или выполнять иные полезные действия.

> **НА ЗАМЕТКУ**
>
> Существуют свойства, доступные только для чтения или только для записи. То есть свойство может быть таким, что можно узнать его значение, но нельзя присвоить значение свойству. А может быть и наоборот: свойству можно присвоить значение, но нельзя узнать значение свойства.

Описывается свойство немного сложнее, чем поле (оно и понятно — в общем случае со свойством связано два метода-аксессора, которые также нужно описать при описании свойства). Вначале указывается тип и название свойства (как и при описании поля), затем в блоке в фигурных скобках описываются методы-аксессоры. Эти методы (их обычно два) описываются специальным образом.

> **НА ЗАМЕТКУ**
>
> Свойство может быть и открытым, и закрытым (при описании свойства можно использовать ключевое слово, определяющее уровень доступа), а также свойство может быть статическим.

Метод, вызываемый при считывании значения свойства (`get`-аксессор), описывается следующим образом: указывается ключевое слово `get`, после которого в фигурных скобках описываются команды, выполняемые при считывании значения свойства. То есть никаких формальных атрибутов метода (тип результата, название, список аргументов) здесь нет. Объяснение простое. Уникальное название методу не нужно, поскольку метод «самостоятельной ценности» не имеет и вызывается при обращении к свойству (то есть имя свойства «замещает» имя метода). Аргументы методу не нужны по определению, поскольку при считывании значения свойства никакие дополнительные значения или атрибуты не используются. А вот результат метод возвращает (хотя идентификатор типа результата явно никак не обозначен). Возвращаемый `get`-аксессором результат — это значение свойства. Поэтому тип результата `get`-аксессора совпадает с типом свойства.

Метод, который вызывается при присваивании значения свойству (`set`-аксессор), описывается таким образом: указывается ключевое слово

set, а в фигурных скобках описываются команды, выполняемые при присваивании значения свойству. Данный метод-аксессор не возвращает результат, и у него нет аргументов. Но один параметр методу все же нужен: это значение, которое присваивается свойству. Поскольку аргумента у метода нет, то аргументом данное значение в метод передать нельзя. Поэтому присваиваемое свойству значение в set-аксессор передается с помощью ключевого слова value. Это ключевое слово, которое в блоке описания set-аксессора отождествляется со значением, присваиваемым свойству.

Ниже представлен шаблон, в соответствии с которым описывается свойство (основные элементы шаблона выделены жирным шрифтом):

```
тип имя{
   get{
      // Код get-аксессора
   }
   set{
      // Код set-аксессора
   }
}
```

При описании свойства можно определить не два, а только один аксессор: или get-аксессор, или set-аксессор. Если для свойства определен только get-аксессор, то значение такого свойства можно прочитать, но такому свойству нельзя присвоить значение. Если для свойства определен только set-аксессор, то значение свойству можно присвоить, но нельзя его прочитать.

НА ЗАМЕТКУ

Далее мы рассмотрим разные способы определения свойств, включая и свойства с одним аксессором. Пока же заметим, что примером свойства, доступного только для чтения, но не доступного для присваивания, может быть свойство Length у массивов и текстовых строк.

Еще одна важная особенность свойства состоит в том, что под свойство место в памяти не выделяется. То есть из того факта, что мы описали свойство, не следует, что при создании объекта под это свойство выделяется память. В этом смысле свойство принципиально отличается

от поля. Поэтому если мы собираемся при работе со свойством запоминать и использовать некоторое значение, то нам придется для этого специально описать поле (обычно закрытое). Фактически свойство — это некая «ширма», за которой может «скрываться» все, что угодно.

НА ЗАМЕТКУ

Поскольку свойство не определяет область памяти и не связано с областью памяти, то свойство нельзя использовать с идентификаторами `ref` и `out`.

В листинге 9.1 представлена программа, в которой есть класс, а в этом классе есть свойство.

Листинг 9.1. Знакомство со свойствами

```
using System;
// Класс со свойством:
class MyClass{
   // Закрытые целочисленные поля:
   private int num;
   private int min;
   private int max;
   // Конструктор с двумя аргументами:
   public MyClass(int a, int b){
      // Присваивание значения полям:
      min=a;
      max=b;
      // Присваивание значения свойству:
      code=(max+min)/2;
   }
   // Описание целочисленного свойства:
   public int code{
      // Метод вызывается при считывании значения свойства:
      get{
         // Значение свойства:
```

```
            return num;
        }
        // Метод вызывается при присваивании
        // значения свойству:
        set{
            // Если присваиваемое значение меньше минимально
            // допустимого:
            if(value<min) num=min;
            // Если присваиваемое значение больше максимально
            // допустимого:
            else if(value>max) num=max;
                 // Если присваиваемое значение попадает в
                 // допустимый диапазон:
                 else num=value;
        }
    }
}
// Класс с главным методом:
class UsingPropsDemo{
    // Главный метод:
    static void Main(){
        // Создание объекта:
        MyClass obj=new MyClass(1,9);
        // Проверка значения свойства:
        Console.WriteLine("Свойство code: "+obj.code);
        // Присваивание значения свойству:
        obj.code=7;
        // Проверка значения свойства:
        Console.WriteLine("Свойство code: "+obj.code);
        // Присваивание значения свойству:
        obj.code=20;
        // Проверка значения свойства:
```

```
        Console.WriteLine("Свойство code: "+obj.code);
        // Присваивание значения свойству:
        obj.code= -10;
        // Проверка значения свойства:
        Console.WriteLine("Свойство code: "+obj.code);
    }
}
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 9.1)

```
Свойство code: 5
Свойство code: 7
Свойство code: 9
Свойство code: 1
```

В программе мы описываем класс с названием `MyClass`. В классе описаны три целочисленных закрытых поля `num`, `min` и `max`. В классе есть конструктор с двумя аргументами (код конструктора мы осудим позже). Но наиболее интересное для нас место в классе — описание свойства с названием `code`. Свойство описано с идентификаторами `public` и `int`, что означает, что свойство открытое и целочисленное. Свойство описано с двумя методами-аксессорами. В теле `get`-аксессора всего одна команда `return num`, которой значением свойства возвращается значение поля `num`.

ПОДРОБНОСТИ

Поскольку свойство `code` описано как такое, что относится к типу `int`, то `get`-аксессор, описанный для этого свойства, должен возвращать значение типа `int`.

При присваивании значения свойству `code` будет вызываться `set`-аксессор, описанный для свойства. В соответствии с кодом этого аксессора выполняются вложенные условные операторы. В этих условных операторах проверяется присваиваемое свойству значение (это значение отождествляется с ключевым словом `value`). Если это значение меньше значения поля `min` (условие `value<min` истинно), то поле `num`

получает значение min (команда num=min). Если присваиваемое свойству значение больше значения поля max (условие value>max истинно), то полю num присваивается значение max (команда num=max). Иначе (если присваиваемое свойству значение попадает в диапазон значений от min до max) полю num присваивается то значение, которое собственно указано справа от оператора присваивания (команда num=value).

ПОДРОБНОСТИ

Мы не объявляли переменную с названием value. Это ключевое слово, обозначающее присваиваемое свойству значение. Вполне разумно отождествлять идентификатор value с некоторой переменной. Причем у переменной есть тип — это тип свойства, которому присваивается значение. В нашем примере value отождествляется с целочисленным значением (тип int).

Что в итоге получается? Получается, что при запросе значения свойства code в действительности возвращается значение поля num. Само поле num закрытое, поэтому вне кода класса значение этому полю напрямую присвоить не удастся. Значение поля num изменяется при присваивании значения свойству code. Алгоритм присваивания не сводится к копированию присваиваемого значения в поле num. Мы хотим, чтобы значение поля num не могло быть меньше значения поля min, а также чтобы оно не могло быть больше значения поля max. Поэтому если присваиваемое свойству code значение попадает в диапазон значений от min до max, то поле num получает то значение, которое реально присваивается. Если же присваиваемое значение выходит за пределы допустимого диапазона значений, то поле num получает минимально/максимально возможное значение (в зависимости от того, за какую границу «выскакивает» присваиваемое значение).

Что касается конструктора класса MyClass, то два его аргумента определяют минимальное и максимальное значение для свойства code (значения полей min и max), а командой code=(max+min)/2 свойству code (а значит, и полю num) присваивается начальное значение.

НА ЗАМЕТКУ

После создания объекта класса MyClass значения полей min и max этого объекта уже изменить нельзя. Значение поля num изменяется при присваивании значения свойству code объекта.

В главном методе программы командой `MyClass obj=new MyClass(1,9)` создается объект `obj` класса `MyClass`. Таким образом, значение свойства `code` объекта `obj` не может выходить за пределы значений от `1` до `9` включительно. Начальное значение свойства `code` равно `5` (поскольку (1+9)/2=5). При выполнении команды `obj.code=7` свойство `code` (поле `num`) получит значение `7`. Выполнение команды `obj.code=20` приводит к тому, что свойство `code` (поле `num`) получит значение `9`. Наконец, выполнение команды `obj.code= -10` приведет к тому, что свойство `code` (поле `num`) получит значение `1`.

НА ЗАМЕТКУ

Подумайте, что произойдет, если при создании объекта класса `MyClass` значение первого аргумента конструктора будет больше значения второго аргумента конструктора. Предложите способ реорганизации программного кода конструктора, чтобы при создании объекта класса `MyClass` значение поля `max` не могло быть меньше значения поля `min`.

Использование свойств

— Куда вы меня несете?
— Навстречу твоему счастью.

из к/ф «Ирония судьбы, или С легким паром»

В этом разделе мы проанализируем несколько примеров, содержащих классы, в которых описываются и используются свойства. Сначала рассмотрим пример, представленный в листинге 9.2.

Листинг 9.2. Использование свойства

```
using System;
// Класс со свойством:
class MyClass{
   // Закрытые целочисленные поля:
   private int first;
   private int last;
   // Конструктор с двумя аргументами:
```

```
public MyClass(int a, int b){
   first=a;   // Значение первого поля
   last=b;    // Значение второго поля
}
// Переопределение метода ToString():
public override string ToString(){
   // Формирование текстового значения:
   string txt="Поля объекта: ";
   txt+=first+" и "+last;
   // Результат метода:
   return txt;
}
// Целочисленное свойство:
public int number{
   // Метод вызывается при считывании значения свойства:
   get{
      // Запоминается значение второго поля:
      int t=last;
      // Новое значение второго поля:
      last=first;
      // Новое значение первого поля:
      first=t;
      // Значение свойства:
      return t;
   }
   // Метод вызывается при присваивании
   // значения свойству:
   set{
      // Новое значение второго поля:
      last=first;
      // Присваивание значения первому полю:
      first=value;
   }
```

```
    }
}
// Класс с главным методом:
class MorePropsDemo{
    // Главный метод:
    static void Main(){
        // Создание объекта:
        MyClass obj=new MyClass(1,9);
        // Проверка значений полей объекта:
        Console.WriteLine(obj);
        // Проверка значения свойства:
        Console.WriteLine("Свойство number: {0}", obj.number);
        // Проверка значений полей объекта:
        Console.WriteLine(obj);
        // Присваивание значения свойству:
        obj.number=5;
        // Проверка значений полей объекта:
        Console.WriteLine(obj);
        // Проверка значения свойства:
        Console.WriteLine("Свойство number: {0}", obj.number);
        // Проверка значений полей объекта:
        Console.WriteLine(obj);
        // Проверка значения свойства:
        Console.WriteLine("Свойство number: {0}", obj.number);
        // Проверка значений полей объекта:
        Console.WriteLine(obj);
        // Присваивание значения свойству:
        obj.number=3;
        // Проверка значений полей объекта:
        Console.WriteLine(obj);
    }
}
```

Ниже представлен результат данной программы.

Результат выполнения программы (из листинга 9.2)

```
Поля объекта: 1 и 9
Свойство number: 9
Поля объекта: 9 и 1
Поля объекта: 5 и 9
Свойство number: 9
Поля объекта: 9 и 5
Свойство number: 5
Поля объекта: 5 и 9
Поля объекта: 3 и 5
```

В программе описан класс `MyClass` с двумя закрытыми полями целочисленного типа `first` и `last`, конструктором с двумя аргументами и свойством `number`. При вызове конструктора аргументы, переданные ему, присваиваются значениями полям объекта. Свойство `number` реализовано таким образом, что мы можем и прочитать значение свойства, и присвоить значение свойству. Но последовательность операций, которые выполняются при считывании значения свойства и при присваивании значения свойству, не самая тривиальная. А именно, если мы считываем значение свойства `number`, то значением возвращается текущее значение поля `last`, но при этом свойства `first` и `last` меняются значениями. Если мы присваиваем значение свойству `number`, то присваиваемое значение записывается в поле `first`, а текущее значение поля `first` присваивается полю `last`.

В главном методе программы командой `MyClass obj=new MyClass(1,9)` создается объект `obj`, поля `first` и `last` которого получают значения `1` и `9` соответственно. Проверка с помощью команды `Console.WriteLine(obj)` подтверждает это.

НА ЗАМЕТКУ

В классе `MyClass` переопределен метод `ToString()`, который результатом возвращает текстовую строку со значениями полей `first` и `last` объекта. Также напомним, что идентификатор `string` является псевдонимом для выражения `System.String`.

При проверке значения свойства number (инструкция obj.number) значение свойства равно 9, а поля first и last обмениваются значениями (новые значения 9 и 1).

При присваивании значения 5 свойству number (команда obj.number=5) поле first получает значение 5, а значение 9 присваивается полю last. Проверка свойства number дает 9, а значения полей объекта станут равны 9 и 5 (при проверке значения свойства поля обмениваются значениями). Если мы еще раз проверим свойство number, его значение будет равно 5, а значения полей станут равны 5 и 9. Далее выполняется команда obj.number=3, вследствие выполнения которой поля получают значения 3 и 5 (поле first получает значение 3, а «старое» значение 5 поля first присваивается значением полю last).

В листинге 9.3 представлена программа, в которой есть класс со свойством, для которого описан только get-аксессор, а set-аксессор не определен. Поэтому такое свойство доступно только для чтения, но не для присваивания значения.

Листинг 9.3. Свойство без set-аксессора

```
using System;
// Класс со свойством:
class MyClass{
   // Открытые целочисленные поля:
   public int first;
   public int last;
   // Конструктор с двумя аргументами:
   public MyClass(int a, int b){
      first=a;   // Значение первого поля
      last=b;    // Значение второго поля
   }
   // Целочисленное свойство (доступно только для чтения):
   public int sum{
      // Метод вызывается при считывании значения свойства:
      get{
         // Значение свойства:
```

```
         return first+last;
      }
   }
}
// Класс с главным методом:
class WithoutSetDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта:
      MyClass obj=new MyClass(1,9);
      // Проверка значения свойства:
      Console.WriteLine("Сумма полей: "+obj.sum);
      // Присваивание значения полю:
      obj.first=6;
      // Проверка значения свойства:
      Console.WriteLine("Сумма полей: "+obj.sum);
      // Присваивание значения полю:
      obj.last=2;
      // Проверка значения свойства:
      Console.WriteLine("Сумма полей: "+obj.sum);
   }
}
```

Результат выполнения такой.

Результат выполнения программы (из листинга 9.3)

```
Сумма полей: 10
Сумма полей: 15
Сумма полей: 8
```

В программе описан класс, у которого имеется два открытых (чтобы проще было менять значения) поля. Свойство `sum`, описанное в классе, имеет только `get`-аксессор. В качестве значения свойства возвращается сумма полей объекта. Если мы меняем значения полей, то меняется и значение свойства. С другой стороны, понятно, что мы не можем

в принципе присвоить значение такому свойству. Здесь, фактически, свойство играет роль, характерную для метода, который вызывается из объекта класса и возвращает результат, вычисляемый на основе значений полей объекта. На практике такие ситуации встречаются достаточно часто.

Более экзотическим является случай, когда свойство доступно для присваивания значения, но не доступно для его считывания. Тем не менее такое тоже возможно. Соответствующее свойство описывается с `set`-аксессором, но без `get`-аксессора. Соответствующий пример представлен в листинге 9.4.

Листинг 9.4. Свойство без get-аксессора

```
using System;
// Класс со свойством:
class MyClass{
   // Закрытое текстовое поле:
   private string code;
   // Конструктор с одним аргументом:
   public MyClass(uint n){
      // Присваивание значения свойству:
      number=n;
   }
   // Переопределение метода ToString():
   public override string ToString(){
      return code; // Результат метода
   }
   // Свойство без get-аксессора:
   public uint number{
      // Метод вызывается при присваивании
      // значения свойству:
      set{
         // Локальная переменная:
         uint num=value;
         // Начальное значение текстового поля:
```

```
         code="";
         // Вычисление битов числа:
         do{
            // Значение последнего (младшего) бита
            // дописывается слева к тексту:
            code=(num%2)+code;
            // Сдвиг кода числа на одну позицию вправо:
            num>>=1;
         }while(num!=0);
      }
   }
}
// Класс с главным методом:
class WithoutGetDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта:
      MyClass obj=new MyClass(5);
      // Бинарный код целого числа:
      Console.WriteLine("Бинарный код числа 5:\t"+obj);
      // Присваивание значения свойству:
      obj.number=45;
      // Бинарный код целого числа:
      Console.WriteLine("Бинарный код числа 45:\t"+obj);
   }
}
```

Ниже приведен результат выполнения программы.

Результат выполнения программы (из листинга 9.4)

```
Бинарный код числа 5:   101
Бинарный код числа 45:  101101
```

В данном случае мы имеем дело с классом `MyClass`, у которого есть свойство `number` типа `uint` (неотрицательное целое число) и закрытое текстовое поле `code`. Свойству можно присвоить значение, но нельзя прочитать значение свойства. При присваивании значения свойству `number` в текстовое поле `code` записывается (в виде текстовой строки) бинарный код этого числа. Метод `ToString()` переопределен в классе так, что результатом метода возвращается значение текстового поля `code`. Конструктору передается аргументом целое неотрицательное число, которое присваивается значением свойству `number`.

В `set`-аксессоре свойства `number` присваиваемое свойству значение записывается в локальную переменную `num` (команда `uint num=value`). Текстовое поле `code` получает начальным значением пустой текст (команда `code=""`). После этого запускается оператор цикла `do`. В теле оператора цикла командой `code=(num%2)+code` к текущему текстовому значению поля `code` слева дописывается значение последнего бита в числовом значении `num`. Для вычисления последнего (младшего) бита в числе (значение переменой `num`) мы вычисляем остаток от деления этого числа на `2` (выражение `num%2`). Остаток от деления на `2` — это число `0` или `1`. После того как значение младшего (на данный момент) бита записано, командой `num>>=1` бинарный код числового значения переменной `num` сдвигается вправо на одну позицию. Младший бит теряется, а новым младшим битом становится тот, который был до этого предпоследним. На следующей итерации этот бит будет прочитан и дописан к текстовому значению поля `code`. Так будет продолжаться до тех пор, пока значение переменной `num` отлично от нуля (условие `num!=0` в инструкции `while`).

ПОДРОБНОСТИ

В `set`-аксессоре поле `code` получает начальное значение `""`. Если свойству `number` присваивается значение `0`, то оператор цикла `do-while` будет выполнен один раз, и в результате текстовое поле `code` получит значение `"0"`.

В главном методе мы создаем объект с передачей значения `5` аргументом конструктору и в итоге получаем бинарный код для этого числа. Затем свойству `number` присваивается новое значение `45`, а в результате приведения объекта к текстовому формату в консоли появляется его бинарный код.

Еще одна программа, иллюстрирующая использование сразу нескольких свойств в одном классе, представлена в листинге 9.5.

Листинг 9.5. Использование различных свойств

```
using System;
// Класс со свойствами:
class MyClass{
   // Закрытое поле-массив:
   private int[] nums;
   // Текстовое свойство без set-аксессора:
   public string content{
      // Метод вызывается при считывании значения свойства:
      get{
         // Если ссылка на массив пустая:
         if(nums==null) return "{}";
         // Формирование текстовой строки:
         string txt="{"+nums[0];
         for(int k=1; k<nums.Length; k++){
            txt+=","+nums[k];
         }
         txt+="}";
         // Значение свойства:
         return txt;
      }
   }
   // Целочисленное свойство без get-аксессора:
   public int element{
      // Метод вызывается при присваивании
      // значения свойству:
      set{
         // Если ссылка на массив пустая:
         if(nums==null){
```

```
            // Создание массива из одного элемента:
            nums=new int[1];
            // Значение единственного элемента массива:
            nums[0]=value;
         }else{// Если ссылка не пустая
            // Создание массива:
            int[] n=new int[nums.Length+1];
            // Заполнение массива:
            for(int k=0; k<nums.Length; k++){
               n[k]=nums[k];
            }
            // Значение последнего элемента в массиве:
            n[nums.Length]=value;
            // Ссылка на созданный массив записывается в
            // поле объекта:
            nums=n;
         }
      }
   }
   // Свойство является ссылкой на массив:
   public int[] data{
      // Метод вызывается при считывании значения свойства:
      get{
         // Создание массива:
         int[] res=new int[nums.Length];
         // Заполнение массива:
         for(int k=0; k<nums.Length; k++){
            res[k]=nums[k];
         }
         // Значение свойства:
         return res;
      }
```

```
      // Метод вызывается при присваивании
      // значения свойству:
      set{
         // Создание массива:
         nums=new int[value.Length];
         // Заполнение массива:
         for(int k=0; k<value.Length; k++){
            nums[k]=value[k];
         }
      }
   }
}
// Класс с главным методом:
class MoreUsingPropsDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта:
      MyClass obj=new MyClass();
      // Проверка содержимого массива из объекта:
      Console.WriteLine(obj.content);
      // Присваивание значения свойству element:
      obj.element=10;
      // Проверка содержимого массива из объекта:
      Console.WriteLine(obj.content);
      // Присваивание значения свойству element:
      obj.element=5;
      obj.element=7;
      // Проверка содержимого массива из объекта:
      Console.WriteLine(obj.content);
      // Считывание значения свойства data:
      int[] A=obj.data;
      // Присваивание значения свойству element:
```

```
      obj.element=12;
      // Отображение содержимого массива A:
      for(int k=0; k<A.Length; k++){
         Console.Write(A[k]+" ");
      }
      Console.WriteLine();
      // Проверка содержимого массива из объекта:
      Console.WriteLine(obj.content);
      // Создание массива:
      int[] B={11,3,6};
      // Присваивание значения свойству data:
      obj.data=B;
      // Изменение значения элемента массива B:
      B[0]=0;
      // Отображение содержимого массива B:
      for(int k=0; k<B.Length; k++){
         Console.Write(B[k]+" ");
      }
      Console.WriteLine();
      // Проверка содержимого массива из объекта:
      Console.WriteLine(obj.content);
   }
}
```

В результате выполнения программы получаем следующее.

Результат выполнения программы (из листинга 9.5)

```
{}
{10}
{10,5,7}
10 5 7
{10,5,7,12}
0 3 6
{11,3,6}
```

Класс MyClass имеет закрытое поле nums, представляющее собой ссылку на целочисленный массив. Этот массив используется при описании свойств content, element и data. Свойство content текстовое, и у этого свойства есть только get-аксессор (свойство можно прочитать, но нельзя присвоить). Свойство возвращает текстовую строку со значениями элементов массива. При вызове get-аксессора для этого свойства в условном операторе проверяется условие nums==null, истинное, если поле nums не ссылается на массив. Если так, то значением свойства возвращается строка "{}".

ПОДРОБНОСТИ

Ключевое слово null означает пустую ссылку. Если переменной ссылочного типа (переменной массива или объектной переменной) не присвоено значение, то значение такой переменной — пустая ссылка (то есть равно null).

Если же поле nums ссылается на массив, то формируется текстовая строка, содержащая значения элементов массива, и эта строка возвращается как значение свойства content.

Целочисленное свойство element не имеет get-аксессора, поэтому свойству можно присвоить значение, но нельзя узнать значение свойства. При присваивании значения свойству в массив nums добавляется новый элемент с присваиваемым значением. Для этого в теле set-аксессора в условном операторе проверяется условие nums==null. Истинность условия означает, что поле nums на массив не ссылается. В таком случае командой nums=new int[1] создается массив из одного элемента, и значение этого элемента есть присваиваемое свойству значение (команда nums[0]=value).

Если условие nums==null в условном операторе ложно, то командой int[] n=new int[nums.Length+1] создается новый массив, размер которого на единицу больше размера массива, на который ссылается поле nums. Сначала выполняется поэлементное копирование значений из массива nums во вновь созданный массив. Остается незаполненным один последний элемент. Командой n[nums.Length]=value этому элементу присваивается значение, которое по факту присваивается свойству element. Наконец, командой nums=n в поле nums записывается ссылка на новый массив. Общий эффект получается такой, что в массиве, на который ссылается поле nums, появился дополнительный последний элемент.

Свойство data является ссылкой на целочисленный массив. Для свойства описаны два аксессора. Это означает, что свойству значением можно присвоить ссылку на массив и результатом свойства является также ссылка на массив.

В get-аксессоре для этого свойства командой int[] res=new int[nums.Length] создается массив такого же размера, что и массив, на который ссылается поле nums. Затем выполняется копирование массивов. Значением свойства является ссылка на новый созданный массив, являющийся копией массива, на который ссылается поле nums.

> **НА ЗАМЕТКУ**
>
> Свойство data значением возвращает ссылку на массив, который является копией массива, на который ссылается поле nums объекта.

В set-аксессоре для свойства data командой nums=new int[value.Length] создается новый массив, и ссылка на него записывается в поле nums. Здесь следует учесть, что ключевое слово value, обозначающее присваиваемое свойству значение, относится к типу int[] (тип свойства data) — то есть является ссылкой на целочисленный массив и обрабатывается как массив. Размер такого массива вычисляется выражением value.Length. С помощью оператора цикла выполняется копирование значений элементов из присваиваемого массива value в созданный массив nums.

В главном методе программы командой MyClass obj=new MyClass() создается объект obj класса MyClass. Его поле nums на массив не ссылается, в чем легко убедиться с помощью команды Console.WriteLine(obj.content). Присваивание значения свойству element командой obj.element=10 приводит к тому, что в массиве nums появляется один элемент. После выполнения команд obj.element=5 и obj.element=7 там появляется еще два элемента.

На следующем этапе командой int[] A=obj.data объявляется переменная массива A и в нее записывается ссылка на копию массива, на который ссылается поле nums объекта obj. Чтобы проверить, что это именно копия, а не сам массив nums, командой obj.element=12 в массив nums объекта obj добавляется еще один элемент. При этом массив A остается неизменным, в чем легко убедиться, сравнив результат отображения содержимого массива A и массива nums из объекта obj.

Далее командой `int[] B={11,3,6}` создается целочисленный массив `B`, и командой `obj.data=B` ссылка на массив присваивается значением свойству `data` объекта `obj`. В результате поле `nums` объекта получает значением ссылку на копию массива `B`. Действительно, проверка показывает, что после выполнения команды `B[0]=0` массив `B` меняется, а массив, на который ссылается поле `nums` объекта `obj`, остается таким, как до выполнения команды `B[0]=0`.

Свойство может быть статическим. С технической точки зрения особенность такого свойства в том, что оно описывается с ключевым словом `static`. Но поскольку свойство не связано с областью памяти, то наличие в классе статического свойства обычно подразумевает, что для реализации этого свойства используются статические поля. Небольшой пример, показывающий, как может быть описано статическое свойство, представлен в листинге 9.6.

Листинг 9.6. Статическое свойство

```
using System;
// Класс со статическим свойством:
class Fibs{
   // Закрытые целочисленные статические поля:
   private static int last=1;
   private static int prev=1;
   // Статическое целочисленное свойство:
   public static int number{
      // Метод вызывается при считывании значения свойства:
      get{
         // Локальная переменная:
         int res=prev;
         // Изменение значения статических полей:
         last=last+prev;
         prev=last-prev;
         // Значение свойства:
         return res;
      }
```

```
      // Метод вызывается при присваивании
      // значения свойству:
      set{
         // Начальное значение статических полей:
         prev=1;
         last=1;
         // Изменение значения статических полей:
         for(int k=2; k<=value; k++){
            last=last+prev;
            prev=last-prev;
         }
      }
   }
}
// Класс с главным методом:
class StaticPropsDemo{
   // Главный метод:
   static void Main(){
      // Отображение значения статического свойства:
      for(int k=1; k<=10; k++){
         Console.Write("{0,4}", Fibs.number);
      }
      Console.WriteLine();
      // Присваивание значения статическому свойству:
      Fibs.number=6;
      // Отображение значения статического свойства:
      for(int k=1; k<=10; k++){
         Console.Write("{0,4}", Fibs.number);
      }
      Console.WriteLine();
      // Присваивание значения статическому свойству:
      Fibs.number=1;
```

```
        // Отображение значения статического свойства:
        for(int k=1; k<=10; k++){
           Console.Write("{0,4}", Fibs.number);
        }
        Console.WriteLine();
    }
}
```

Результат выполнения программы следующий.

Результат выполнения программы (из листинга 9.6)

```
   1   1   2   3   5   8  13  21  34  55
   8  13  21  34  55  89 144 233 377 610
   1   1   2   3   5   8  13  21  34  55
```

Мы описали класс `Fibs` со статическим свойством `number`. При проверке значения свойства результатом оно возвращает число из последовательности Фибоначчи. Причем каждый раз это число новое.

НА ЗАМЕТКУ

В последовательности Фибоначчи два первых числа равны `1`, а каждое следующее число — это сумма двух предыдущих чисел. Получается последовательность чисел `1`, `1`, `2`, `3`, `5`, `8`, `13`, `21`, `34`, `55` и так далее.

При присваивании значения свойству `number` число, которое присваивается свойству, определяет порядковый номер числа из последовательности Фибоначчи, которое будет возвращаться в качестве значения свойства. Например, если мы присваиваем свойству `number` значение `6`, то это означает, что при считывании значения свойства мы получим значение `8`, поскольку именно число `8` является шестым по счету в последовательности Фибоначчи. Если далее мы еще раз прочитаем значение свойства `number`, то получим значение `13`, а затем при считывании значения свойства получим значение `21` и так далее.

Для реализации такого подхода мы описываем два закрытых целочисленных статических поля `last` и `prev` с начальными единичными значениями. Эти поля нам нужны для того, чтобы записать в них значения двух последних чисел из последовательности Фибоначчи. Нам нужно

именно два числа, поскольку следующее число в последовательности вычисляется как сумма двух предыдущих. Начальные значения полей объясняются тем, что первые два числа в последовательности — это 1 и 1.

Свойство number описано с ключевым словом static. В get-аксессоре свойства объявляется локальная переменная res, в которую записывается значение поля prev. Затем выполняются две команды last=last+prev и prev=last-prev, которыми полю last значением присваивается новое число Фибоначчи, а полю prev значением присваивается то число, которое раньше было записано в поле last. После этого значение переменной res (то значение, которое в самом начале было записано в поле prev) возвращается как значение свойства. Таким образом, получается, что при считывании значения свойства number результатом возвращается значение поля prev, а затем вычисляется новое число Фибоначчи и обновляются значения полей last и prev так, что поле last содержит значение нового вычисленного числа, а в поле prev записывается значение предыдущего числа.

ПОДРОБНОСТИ

Допустим, поля prev и last содержат два числа из последовательности Фибоначчи. Наша задача — сделать «шаг вперед»: вычислить новое число Фибоначчи и записать в поля новую пару чисел. Чтобы понять алгоритм вычислений, обозначим исходное значение поля prev как Δ, а исходное значение поля last обозначим как □. Необходимо сделать так, чтобы в поле last было записано значение Δ+□, а в поле prev должно быть записано значение □. После выполнения команды last=last+prev в поле last записывается значение Δ+□. Поле prev остается со значением Δ. После выполнения команды prev=last-prev поле prev получает значение □, что нам и нужно.

В set-аксессоре для свойства number командами prev=1 и last=1 полям prev и last присваиваются единичные значения. После этого запускается оператор цикла, в котором индексная переменная k принимает значения от 2 до присваиваемого свойству значения value. За один цикл выполняются команды last=last+prev и prev=last-prev. Каждый раз, когда выполняются эти команды (мы их анализировали выше), вычисляется новое число Фибоначчи. С учетом того, что начальные единичные значения полей prev и last соответствуют первой паре чисел в последовательности, а первое значение индексной переменной k в операторе цикла равно 2, то после выполнения оператора цикла в поле prev

будет записано число из последовательности Фибоначчи, порядковый номер которого определяется значением `value`, присваиваемым свойству. В поле `last` будет записано следующее число в последовательности.

В главном методе программы запускается оператор цикла, в котором выполняется `10` циклов. За каждый цикл в консольном окне отображается значение, возвращаемое инструкцией `Fibs.number`.

ПОДРОБНОСТИ

В теле оператора цикла выполняется команда `Console.Write("{0,4}", Fibs.number)`. Инструкция `{0,4}` означает, что отображается значение выражения `Fibs.number` и под это значение (число) отводится `4` позиции.

В результате в консольном окне отображаются первые `10` чисел из последовательности Фибоначчи. Затем командой `Fibs.number=6` присваивается значение свойству `number`. Затем снова выполняется оператор цикла и `10` раз подряд отображается значение статического свойства `number`. В итоге получаем еще одну строку из `10` чисел Фибоначчи, но на этот раз числа отображаются, начиная с шестого числа. Наконец, после выполнения команды `Fibs.number=1` и выполнения оператора цикла получаем строку из `10` чисел Фибоначчи, начиная с первого числа в последовательности.

Знакомство с индексаторами

> Братцы, кончайте философствовать, он сейчас возникнет уже.
>
> *из к/ф «Кин-дза-дза»*

Теперь пришло время познакомиться с *индексаторами*. Индексатор является специальным членом класса, наличие которого в классе позволяет индексировать объекты класса. Когда мы говорим об индексировании объекта, то подразумеваем, что после имени объекта в квадратных скобках указывается индекс (или индексы) — конечно, при условии, что такое выражение имеет смысл. Таким образом, описав в классе индексатор, мы получаем возможность обращаться с объектом класса так, как если бы этот объект был массивом. Это очень важная и перспективная возможность.

Прежде чем приступить к изучению вопроса о том, как описывается индексатор, остановимся на том, что происходит при индексировании объекта (в случае, когда в соответствующем классе описан индексатор). Итак, представим, что имеется объект, который индексируется — после имени объекта в квадратных скобках указан индекс (для простоты будем полагать, что индекс один). Есть две ситуации, когда такая инструкция может быть использована. В одном случае мы хотим узнать (прочитать) значение данного выражения. В другом случае такому выражению может присваиваться значение. И та и другая ситуация разрешается способом, подобным к тому, как это делается для свойства, а именно — с каждый индексатором связано два специальных метода-аксессора. При считывании значения выражения вызывается `get`-аксессор. При присваивании значения выражению вызывается `set`-аксессор. Единственное принципиальное отличие от свойства состоит в том, что в случае с индексатором в операциях считывания значения и присваивания значения участвует еще и индекс.

Описывается индексатор следующим образом. Кроме спецификатора уровня доступа (обычно это ключевое слово `public`), указывается ключевое слово, определяющее тип значения, возвращаемого при индексировании объекта. Затем указывается ключевое слово `this`. После ключевого слова `this` в квадратных скобках указывается тип индекса и его формальное обозначение. После этого в блоке из фигурных скобок описываются аксессоры. В общем случае их два (хотя может быть и индексатор только с одним аксессором): это `get`-аксессор, который вызывается при считывании значения выражения с проиндексированным объектом, и `set`-аксессор, который вызывается при присваивании значения выражению с проиндексированным объектом. Аксессоры описываются в принципе так же, как и для свойства, но кроме ключевого слова `value`, определяющего присваиваемое значение в `set`-аксессоре, в обоих аксессорах еще используется и переменная, обозначающая индекс. Ниже представлен шаблон для описания индексатора (жирным шрифтом выделены основные элементы шаблона):

```
тип_индексатора this[тип_индекса индекс]{
   // Метод вызывается при считывании значения выражения
   // с проиндексированным объектом:
   get{
      // Код get-аксессора
   }
```

```
   // Метод вызывается при присваивании значения выражению
   // с проиндексированным объектом:
   set{
      // Код set-аксессора
   }
}
```

Часто индексатор описывается с целочисленным индексом (тип индекса указывается как `int`). Но это не обязательно — тип индекса может быть и другим. Более того, индексатор можно *перегружать* — в классе может быть описано несколько индексаторов, которые отличаются количеством и/или типом индексов. Вместе с тем индексатор не может быть статическим, а также индексатор нельзя использовать с идентификаторами `ref` и `out`.

НА ЗАМЕТКУ

Напомним, что ключевое слово `this`, используемое в описании индексатора, может иметь и иное назначение. Так, в программном коде класса, в описании методов (в том числе аксессоров, конструкторов и деструкторов) ключевое слово `this` может использоваться для обозначения объекта, из которого вызывается метод.

В листинге 9.7 представлена программа, в которой есть класс, а в этом классе описан индексатор.

Листинг 9.7. Знакомство с индексаторами

```
using System;
// Класс с индексатором:
class MyClass{
   // Закрытое поле, являющееся ссылкой на массив:
   private int[] nums;
   // Конструктор с целочисленным аргументом:
   public MyClass(int n){
      // Создание массива:
      nums=new int[n];
      // Заполнение массива:
      for(int k=0; k<nums.Length; k++){
```

```
            nums[k]=0;
        }
    }
    // Переопределение метода ToString():
    public override string ToString(){
        // Формирование текстовой строки:
        string txt="{"+nums[0];
        for(int k=1; k<nums.Length; k++){
            txt+=","+nums[k];
        }
        txt+="}";
        // Результат метода:
        return txt;
    }
    // Целочисленное свойство:
    public int length{
        // Метод вызывается при считывании значения свойства:
        get{
            // Значение свойства:
            return nums.Length;
        }
    }
    // Целочисленный индексатор с целочисленным индексом:
    public int this[int k]{
        // Метод вызывается при считывании значения
        // объекта с индексом:
        get{
            // Значение выражения:
            return nums[k];
        }
        // Метод вызывается при присваивании значения
        // объекту с индексом:
```

```
      set{
         // Присваивание значения элементу массива:
         nums[k]=value;
      }
   }
}
// Класс с главным методом:
class UsingIndexerDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта:
      MyClass obj=new MyClass(5);
      // Отображение содержимого массива из объекта:
      Console.WriteLine(obj);
      // Присваивание значений элементам массива из объекта
      // с использованием индексирования объекта:
      for(int k=0; k<obj.length; k++){
         // Используется индексирование объекта:
         obj[k]=2*k+1;
      }
      // Отображение содержимого массива из объекта:
      Console.WriteLine(obj);
      // Поэлементное отображение массива из объекта
      // с использованием индексирования объекта:
      for(int k=0; k<obj.length; k++){
         // Используется индексирование объекта:
         Console.Write(" "+obj[k]);
      }
      Console.WriteLine();
   }
}
```

При выполнении программы получаем такой результат.

Результат выполнения программы (из листинга 9.7)

```
{0,0,0,0,0}
{1,3,5,7,9}
 1 3 5 7 9
```

Описывая класс `MyClass`, мы предусмотрели в нем закрытое поле `nums`, представляющее собой ссылку на целочисленный массив. В классе переопределен метод `ToString()`, так что результатом метода возвращается текстовая строка со значениями элементов массива. Конструктору класса передается один целочисленный аргумент, определяющий размер массива, на который ссылается поле `nums`. Массив создается при выполнении кода конструктора. Также все элементы массива получают нулевое значение.

НА ЗАМЕТКУ

Строго говоря, при создании массива его элементы автоматически получают нулевые значения. Тем не менее мы в конструкторе добавили блок кода (оператор цикла), которым элементам массива присваиваются начальные значения.

В классе есть целочисленное свойство `length`, у которого имеется только `get`-аксессор. Значением свойства `length` является размер массива, на который ссылается поле `nums` (определяется выражением `nums.Length`). Понятно, что присвоить значение свойству `length` нельзя.

Описание целочисленного индексатора с целочисленным индексом начинается с ключевого слова `public`. Идентификатор `int` означает, что результатом выражения, в котором есть объект с индексом, является целое число. Также целое число можно присвоить значением такому выражению.

НА ЗАМЕТКУ

Возможность прочитать значение или присвоить значение выражению на основе проиндексированного объекта определяется наличием или отсутствием соответствующего аксессора.

После обязательного ключевого слова `this` в квадратных скобках указана инструкция `int k`. Она определяет тип и имя индекса. То есть в программном коде аксессоров для этого индексатора под `k` следует подразумевать

индекс, указанный в квадратных скобках после имени объекта. И этот индекс является целочисленным (тип `int`). В теле `get`-аксессора всего одна команда `return nums[k]`. Таким образом, если после имени объекта в квадратных скобках указать индекс, то значением такого выражения является значение элемента массива `nums` с таким же индексом.

В `set`-аксессоре выполняется команда `nums[k]=value`. Ключевое слово `value` отождествляется с присваиваемым значением. По факту оно присваивается элементу массива `nums` с индексом `k`. Получается, что когда мы присваиваем значение объекту с индексом, то в действительности такое значение получит элемент с таким же индексом в массиве объекта.

В главном методе программы мы проверяем работу индексатора. Для этого мы командой `MyClass obj=new MyClass(5)` создаем объект `obj` с массивом из пяти элементов. Вначале этот массив заполнен нулями (проверяем это командой `Console.WriteLine(obj)`). Затем выполняется оператор цикла, в котором перебираются элементы массива в объекте `obj`. Размер массива вычисляется выражением `obj.length`. За каждый цикл (при фиксированном индексе `k`) командой `obj[k]=2*k+1` соответствующему элементу массива присваивается значение. После того как мы проверяем содержимое массива в объекте, с помощью еще одного оператора цикла мы поэлементно выводим значения элементов массива в консоль. При этом используется индексирование объекта `obj` (инструкция вида `obj[k]`).

Использование индексаторов

Зря старались! Бриллиантов там нет.

из к/ф «Бриллиантовая рука»

Теперь мы рассмотрим несколько примеров, в которых различными способами описываются индексаторы. Во всех примерах речь будет идти об индексаторах с одним индексом. Начнем с программы в листинге 9.8, которая наглядно иллюстрирует, что совсем необязательно, чтобы за индексатором «прятался» массив.

Листинг 9.8. Использование индексатора

```
using System;
// Класс с индексатором:
```

```
class MyClass{
   // Закрытое целочисленное поле:
   private int code;
   // Конструктор с символьным аргументом:
   public MyClass(char s){
      // Присваивание значения полю:
      code=s;
   }
   // Символьный индексатор с целочисленным индексом:
   public char this[int k]{
      // Метод вызывается при считывании значения выражения
      // с проиндексированным объектом:
      get{
         // Значение свойства:
         return (char)(code+k);
      }
      // Метод вызывается при присваивании значения
      // выражению с проиндексированным объектом:
      set{
         // Присваивание значения полю:
         code=value-k;
      }
   }
}
// Класс с главным методом:
class MoreIndexerDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта:
      MyClass obj=new MyClass('A');
      // Индексирование объекта для считывания значения:
      for(int k=0; k<10; k++){
```

```
            Console.Write(obj[k]+" ");
        }
        Console.WriteLine();
        // Присваивание значения выражению с
        // индексированным объектом:
        obj[5]='Q';
        // Индексирования объекта для считывания значения:
        for(int k=0; k<10; k++){
            Console.Write(obj[k]+" ");
        }
        Console.WriteLine();
        // Использование отрицательного индекса
        // при индексировании объекта:
        for(int k=0; k<10; k++){
            Console.Write(obj[-k]+" ");
        }
        Console.WriteLine();
    }
}
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 9.8)

```
A B C D E F G H I J
L M N O P Q R S T U
L K J I H G F E D C
```

Здесь в классе `MyClass` есть закрытое целочисленное поле `code`, значение которому присваивается при создании объекта. Причем аргументом конструктору передается символьное значение. Поэтому в поле `code` записывается код символа, переданного аргументом конструктору.

Индексатор описан как такой, что относится к типу `char` (тип присваиваемого и возвращаемого значения), а индекс (обозначен как `k`) имеет тип `int`. В `get`-аксессоре индикатора выполняется команда `return`

(char)(code+k), в соответствии с которой значением индексатора возвращается символ, код которого получается прибавлением к значению поля code значения индекса, указанного в квадратных скобках после имени объекта.

В set-аксессоре выполняется команда code=value-k, которой значением полю code присваивается разность значения, указанного в операции присваивания, и индекса, указанного в выражении, которому присваивается значение. Причем подчеркнем, что параметр value в данном случае является символьным. Поэтому получается, что в поле code записывается код символа, который в кодовой таблице находится на k позиций перед тем символом, который указан в команде присваивания.

В главном методе командой MyClass obj=new MyClass('A') создается объект obj. Аргументом конструктору передан символ 'A'. Поэтому значением выражения obj[k] при заданном значении индекса k является символ, код которого на k больше, чем код символа 'A'. При выполнении оператора цикла, в котором индексная переменная k пробегает значения от 0 до 9 включительно, отображается значение выражения obj[k], в результате чего появляется строка из десяти символов, начиная с символа 'A' и заканчивая символом 'J'.

При выполнении команды obj[5]='Q' в поле code объекта obj записывается код символа, находящегося в кодовой таблице за 5 позиций по отношению к символу 'Q'. Это символ 'L', и его код записывается в поле code. Поэтому при выполнении оператора цикла, в котором индексная переменная k принимает значения от 0 до 9 и за каждый цикл отображается значение выражения obj[k], мы получаем последовательность из десяти символов, начиная с символа 'L' и заканчивая символом 'U'.

Наконец, пикантность ситуации в том, что формально нам никто не запрещает использовать при индексации объекта отрицательный индекс. Главное, чтобы код аксессоров имел при этом смысл. Если мы выполним еще один оператор цикла, но отображать будем значение выражения obj[-k], то получим последовательность символов, но в обратном порядке: от символа 'L' до символа 'C' включительно.

В следующем примере рассматривается ситуация, когда индексатор имеет только get-аксессор. Интересующая нас программа представлена в листинге 9.9.

Листинг 9.9. Индексатор без set-аксессора

```
using System;
// Класс с индексатором:
class MyClass{
   // Закрытое целочисленное поле:
   private int number;
   // Конструктор с одним аргументом:
   public MyClass(int n){
      // Присваивание значения полю:
      number=n;
   }
   // Целочисленный индексатор с целочисленным индексом:
   public int this[int k]{
      // Метод вызывается при считывании значения выражения
      // с проиндексированным объектом:
      get{
         // Целочисленная переменная:
         int n=number;
         // "Отбрасывание" цифр из младших разрядов:
         for(int i=1; i<k; i++){
            n/=10;
         }
         // Значение свойства:
         return n%10;
      }
   }
}
// Класс с главным методом:
class WithoutSetIndexerDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта:
```

```
      MyClass obj=new MyClass(12345);
      // Цифры в десятичном представлении числа:
      for(int k=1; k<9; k++){
         Console.Write(" | "+obj[k]);
      }
      Console.WriteLine(" |");
   }
}
```

Результат выполнения программы вполне ожидаем.

Результат выполнения программы (из листинга 9.9)

```
| 5 | 4 | 3 | 2 | 1 | 0 | 0 | 0 |
```

У класса `MyClass` есть закрытое целочисленное поле, значение которому присваивается при создании объекта. Индексатор, описанный в классе, имеет только `get`-аксессор. При индексировании объекта результатом возвращается цифра в десятичном представлении числа, записанного в поле `number`. Индекс, указанный при индексировании объекта, определяет позицию цифры в десятичном представлении числа.

НА ЗАМЕТКУ

Индексатор определен так, что наименьшему разряду (единице) в десятичном представлении числа соответствует индекс `1`. Десяткам соответствует индекс `2`, сотням соответствует индекс `3` и так далее. Если при индексировании объекта указать индекс, меньший `1`, то результатом будет цифра в разряде единиц. То есть для отрицательного индекса или индекса `0` результат такой же, как и для индекса `1`.

Для вычисления значения индексатора в теле `get`-аксессора командой `int n=number` объявляется локальная переменная `n`, в которую записывается значение поля `number`. Задача состоит в том, чтобы вычислить цифру, которая находится в десятичном представлении числа в позиции с порядковым номером `k` (целочисленный индекс, указанный в описании индексатора). Для этого мы «отбрасываем» в десятичном представлении числа `k-1` цифру, а цифра, оказавшаяся последней, возвращается как результат. Для этого сначала запускается оператор цикла, и в нем

k-1 раз выполняется команда n/=10, которой текущее значение переменной n делится нацело на 10, а результат записывается в переменную n. Одна такая операция эквивалентна отбрасыванию в десятичном представлении значения переменной n одной цифры в младшем разряде (отбрасывается цифра справа). После выполнения оператора цикла интересующая нас цифра станет последней. Ее можно вычислить как остаток от деления числа на 10 (команда n%10). Это и есть результат.

В главном методе программы мы создаем объект obj, полю number которого присваивается значение 12345. Затем запускается оператор цикла (из восьми циклов), и за каждый цикл отображается значение obj[k]. В итоге мы получаем цифры из десятичного представления числа, только не справа налево (как они записаны в числе), а слева направо. Причем нули соответствуют тем разрядам, которые в числе не представлены.

Еще один пример индексатора с одним аксессором представлен в следующей программе. Там у индексатора есть только set-аксессор. Рассмотрим программный код в листинге 9.10.

Листинг 9.10. Индексатор без get-аксессора

```
using System;
// Класс с индексатором:
class MyString{
   // Закрытое текстовое поле:
   private string text;
   // Конструктор с текстовым аргументом:
   public MyString(string t){
      text=t;
   }
   // Операторный метод для неявного преобразования
   // текстового значения в объект класса MyString:
   public static implicit operator MyString(string t){
      return new MyString(t);
   }
   // Переопределение метода ToString():
   public override string ToString(){
      return text;
```

```
   }
   // Символьный индексатор с целочисленным индексом:
   public char this[int k]{
      // Метод вызывается при присваивании значения
      // выражению с индексированным объектом:
      set{
         // Проверка значения индекса:
         if(k<0||k>=text.Length) return;
         // Текстовая переменная:
         string t="";
         // Добавление символов к тексту:
         for(int i=0; i<k; i++){
            t+=text[i];
         }
         // Добавление в текст присваиваемого символа:
         t+=value;
         // Добавление символов к тексту:
         for(int i=k+1; i<text.Length; i++){
            t+=text[i];
         }
         // Новое значение текстового поля:
         text=t;
      }
   }
}
// Класс с главным методом:
class WithoutGetIndexerDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта:
      MyString txt="Муха";
      // Проверка текста:
      Console.WriteLine(txt);
```

```
        // Попытка изменить символ в тексте:
        txt[-1]='Ы';
        // Проверка текста:
        Console.WriteLine(txt);
        // Попытка изменить символ в тексте:
        txt[4]='Ъ';
        // Проверка текста:
        Console.WriteLine(txt);
        // Изменение символа в тексте:
        txt[0]='С';
        // Проверка текста:
        Console.WriteLine(txt);
        // Изменение символа в тексте:
        txt[1]='л';
        // Проверка текста:
        Console.WriteLine(txt);
        // Изменение символа в тексте:
        txt[2]='о';
        // Проверка текста:
        Console.WriteLine(txt);
        // Изменение символа в тексте:
        txt[3]='н';
        // Проверка текста:
        Console.WriteLine(txt);
    }
}
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 9.10)

```
Муха
Муха
Муха
Суха
```

```
Слха
Слоа
Слон
```

В этой программе мы создаем очень скромное подобие класса, предназначенного для работы с текстом. Описанный нами класс с индексатором называется `MyString`. У класса есть закрытое текстовое поле `text` и конструктор с текстовым аргументом (переданный конструктору аргумент присваивается значением текстовому полю объекта). Также мы описали операторный метод для неявного приведения типа `string` в тип `MyString`. Благодаря этому объектным переменным класса `MyString` можно присваивать текстовые значения. В результате будет создаваться новый объект класса `MyString`, а текстовое поле этого объекта определяется тем текстовым значением, которое присваивается объектной переменной.

Еще в классе `MyString` переопределен метод `ToString()`. Переопределен он таким образом, что результатом возвращается текст из поля `text`.

Индексатор, описанный в классе `MyString`, имеет целочисленный индекс, а в нем есть только `set`-аксессор. Поэтому проиндексированному объекту можно только присвоить значение. Поскольку индексатор описан как относящийся к типу `char`, то и присваиваться должно символьное значение.

При присваивании символьного значения выражению с проиндексированным объектом создается иллюзия того, что в текстовом поле объекта меняется значение символа с соответствующим индексом. Делается это так. Сначала с помощью условного оператора проверяется значение индекса `k`. Условие `k<0||k>=text.Length` истинно в том случае, если индекс меньше нуля или больше максимально допустимого индекса в тексте (определяется длиной текста). В этом случае выполняется инструкция `return`, которая завершает работу аксессора, и с текстовым полем объекта, соответственно, ничего не происходит. Если же этого не происходит, то индекс попадает в допустимый диапазон значений и начинается процесс вычисления нового текстового значения для поля `text`. Для этого объявляется локальная текстовая переменная `t` с пустой текстовой строкой в качестве начального значения. Затем запускается оператор цикла, в котором к текстовой строке последовательно дописываются начальные символы из поля `text`, но только до символа с индексом `k` (этот символ не копируется). Далее командой `t+=value` к текстовой строке дописывается тот символ, который присваивается значением выражению с проиндексированным объектом. То есть

получается, что вместо символа с индексом k из текстового поля text в текстовую строку дописывается тот символ, что указан в операции присваивания (определяется значением параметра value). Затем снова запускается оператор цикла, с помощью которого к текстовой строке из переменной t дописываются все оставшиеся символы из текстового поля text. По завершении этого оператора цикла командой text=t текстовое поле получает новое значение. Новое значение поля text отличается от предыдущего значения одним символом.

В главном методе программы командой MyString txt="Муха" создается объект txt класса MyString (здесь использована операция неявного приведения типов). Также главный метод содержит примеры использования индексатора с индексом, значение которого выходит за допустимый диапазон (команды txt[-1]='Ы' и txt[4]='Ъ'). Есть также команды (txt[0]='С', txt[1]='л', txt[2]='о' и txt[3]='н'), в которых значение индекса корректно. Для проверки значения текстового поля объекта txt используется переопределение метода ToString() (метод вызывается, когда объект txt передается аргументом методу WriteLine()).

В следующем примере описывается класс с индексатором, индекс которого не является целым числом. Соответствующая программа представлена в листинге 9.11.

Листинг 9.11. Индексатор с нечисловым индексом

```
using System;
// Класс с индексатором:
class MyClass{
   // Целочисленное поле:
   public int code;
   // Конструктор с одним аргументом:
   public MyClass(int n){
      code=n;
   }
   // Целочисленный индексатор с индексом, который является
   // объектом класса MyClass:
   public int this[MyClass obj]{
      // Метод вызывается при считывании значения
```

```
      // выражения с проиндексированным объектом:
      get{
         // Результат:
         return code-obj.code;
      }
      // Метод вызывается при присваивании значения
      // выражению с проиндексированным объектом:
      set{
         // Присваивается значение полю:
         code=obj.code+value;
      }
   }
}
// Класс с главным методом:
class NonIntIndexDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта:
      MyClass A=new MyClass(100);
      // Проверка значения поля объекта:
      Console.WriteLine("Объект A: {0}", A.code);
      // Создание объекта:
      MyClass B=new MyClass(150);
      // Проверка значения поля объекта:
      Console.WriteLine("Объект B: {0}", B.code);
      // Использование индексатора:
      int num=A[B];
      Console.WriteLine("Выражение A[B]={0}", num);
      Console.WriteLine("Выражение B[A]={0}", B[A]);
      A[B]=200;
      // Проверка значения поля объекта:
      Console.WriteLine("Объект A: {0}", A.code);
   }
}
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 9.11)

```
Объект A: 100
Объект B: 150
Выражение A[B] = -50
Выражение B[A] = 50
Объект A: 350
```

Здесь мы описываем класс `MyClass`, у которого есть открытое целочисленное поле `code`, конструктор с одним аргументом и индексатор. Тип индексатора определяется ключевым словом `int`, а вот индекс в индексаторе описан как `MyClass obj`. Аксессоры индексатора описаны так, что результатом выражения вида `A[B]`, в котором `A` и `B` являются объектами класса `MyClass`, является разность значений поля `code` объекта `A` и объекта `B`. Например, если поле `code` объекта `A` равно `100`, а поле `code` объекта `B` равно `150`, то результатом выражения `A[B]` будет `–50` (разность значений `100` и `150`). Значение выражения `B[A]` при этом равно `50` (разность значений `150` и `100`). Если выражению вида `A[B]` присваивается целочисленное значение, то поле `code` объекта `A` получит новое значение, которое равно сумме значения поля `code` объекта `B` и значения, присваиваемого выражению `A[B]`. Так, при значении поля `code` объекта `B` равном `150` в результате выполнения команды `A[B]=200` поле `code` объекта `A` получит значение `350` (сумма значений `150` и `200`).

Двумерные индексаторы

> Дело государственной важности. Возможна погоня.
>
> *из к/ф «Бриллиантовая рука»*

Двумерный индексатор описывается, в принципе, так же, как и одномерный индексатор. Отличие лишь в том, что теперь в индексаторе описывается два индекса (могут быть разного типа). Для каждого индекса указывается тип, описания индексов в квадратных скобках разделяются запятыми. При индексировании объектов также указывается два индекса. В листинге 9.12 представлена программа, дающая представление о том, как описывается и используется двумерный индексатор.

Листинг 9.12. Знакомство с двумерными индексаторами

```
using System;
// Класс с двумерным индексатором:
class MyClass{
   // Закрытое поле, являющееся ссылкой на двумерный
   // символьный массив:
   private char[,] symbs;
   // Конструктор с двумя аргументами:
   public MyClass(int a, int b){
      // Создание двумерного массива:
      symbs=new char[a, b];
      // Заполнение двумерного массива.
      // Перебор строк массива:
      for(int i=0; i<symbs.GetLength(0); i++){
         // Перебор столбцов массива:
         for(int j=0; j<symbs.GetLength(1); j++){
            symbs[i, j]='0'; // Значение элемента массива
         }
      }
   }
   // Метод для отображения содержимого массива:
   public void show(){
      Console.WriteLine("Двумерный массив:");
      // Перебор строк массива:
      for(int i=0; i<symbs.GetLength(0); i++){
         // Перебор столбцов массива:
         for(int j=0; j<symbs.GetLength(1); j++){
            // Отображение значения элемента:
            Console.Write(symbs[i, j]+" ");
         }
         Console.WriteLine();
      }
```

```
   }
   // Двумерный индексатор:
   public char this[int i, int j]{
      // Метод вызывается при считывании значения
      // выражения с проиндексированным объектом:
      get{
         // Результат:
         return symbs[i, j];
      }
      // Метод вызывается при присваивании значения
      // выражению с проиндексированным объектом:
      set{
         // Значение элемента массива:
         symbs[i, j]=value;
      }
   }
}
// Класс с главным методом:
class TwoDimIndexerDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта:
      MyClass obj=new MyClass(2,3);
      // Проверка содержимого массива:
      obj.show();
      // Индексирование объекта:
      obj[0,0]='A';
      obj[1,2]='Z';
      // Проверка содержимого массива:
      obj.show();
      Console.WriteLine("Проверка:");
      // Индексирование объекта:
```

```
        Console.WriteLine("obj[0,0]={0}", obj[0,0]);
        Console.WriteLine("obj[1,1]={0}", obj[1,1]);
        Console.WriteLine("obj[1,2]={0}", obj[1,2]);
    }
}
```

При выполнении программы получаем такой результат.

Результат выполнения программы (из листинга 9.12)

```
Двумерный массив:
0 0 0
0 0 0
Двумерный массив:
A 0 0
0 0 Z
Проверка:
obj[0,0]=A
obj[1,1]=0
obj[1,2]=Z
```

В этой программе мы описываем класс `MyClass`, в котором есть закрытое поле `symbs`, представляющее собой ссылку на двумерный символьный массив. Сам массив создается при вызове конструктора, которому передается два целочисленных аргумента. Они определяют размеры массива. Все элементы массива получают символьное значение `'0'`. Также в классе описан метод `show()`, при вызове которого отображается содержимое двумерного массива, связанного с объектом, из которого вызывается метод.

Двумерный индексатор описан в классе `MyClass` с ключевым словом `char`, обозначающим тип возвращаемого и присваиваемого значения. У индексатора два индекса — оба целочисленные. При считывании значения выражения вида `obj[i, j]` (где `obj` является объектом класса `MyClass`) результатом возвращается значение соответствующего элемента массива `symbs[i, j]`, а при присваивании значения выражению вида `obj[i, j]` значение в действительности присваивается элементу массива `symbs[i, j]`. Выполнение подобных операций проиллюстрировано в главном методе программы.

Еще один пример использования двумерного индексатора представлен в программе в листинге 9.13. В данном примере описывается класс `MyClass`, в котором имеется три массива (предполагается, что одного и того же размера): символьный, текстовый и целочисленный. С помощью данного класса мы в очень простом варианте реализуем конструкцию, напоминающую *ассоциативный контейнер*. В таком контейнере есть *элементы*, у каждого элемента имеется *значение*, а доступ к элементу осуществляется на основе *ключей*. Ключ во многом схож с индексом, но в отличие от индекса ключ не обязательно должен быть целочисленным. Мы будем исходить из того, что ключи элементов записываются в символьный и текстовый массивы, а значение элемента хранится в целочисленном массиве.

ПОДРОБНОСТИ

Термин «элемент» в данном примере используется в двух смыслах. Во-первых, мы условно полагаем, что объект класса содержит некоторые «виртуальные» элементы, которые имеют значение (целочисленные) и доступ к которым осуществляется по двум ключам (символ и текст). Во-вторых, есть три массива: символьный, текстовый и целочисленный. Массивы одинакового размера. Между элементами массивов существует строгое соответствие: если зафиксировать некоторый целочисленный индекс, то элементы в символьном и текстовом массивах с этим индексом определяют значения ключей нашего «виртуального» элемента, а элемент в целочисленном массиве с таким же индексом определяет значение нашего «виртуального» элемента.

С помощью индексатора мы реализуем две операции: считывание значения выражения на основе объекта с двумя индексами (символ и текст) и присваивание значения такому выражению. При считывании значения в символьном и текстовом массивах объекта выполняется поиск элементов, значения которых совпадают со значениями индексов. Причем совпадение должно быть «двойное»: значение элемента в символьном массиве должно совпадать со значением символьного индекса, а значение элемента на такой же позиции в текстовом массиве должно совпадать со значением текстового индекса. Если такое совпадение есть, то значением выражения возвращается значение соответствующего элемента из целочисленного массива. Если совпадение не найдено, то в каждый из трех массивов (символьный, текстовый и целочисленный) добавляется новый элемент. Значение добавляемого элемента

такое: для символьного и текстового массивов это значения индексов, а для целочисленного массива значение добавляемого элемента равно 0.

При присваивании значения выражению (с проиндексированным объектом) выполняется поиск по индексам. Если элемент найден, то ему присваивается новое значение (точнее, в целочисленный массив для соответствующего элемента записывается новое значение). Если элемента с указанными индексами в объекте нет, то он добавляется в объект (каждый из трех массивов получает по одному новому элементу).

ПОДРОБНОСТИ

Операцию добавления нового элемента в массив следует понимать в том смысле, что создается новый массив, размер которого на единицу больше, чем размер исходного массива. В новый массив копируются значения элементов из исходного массива, а последний элемент получает значение того элемента, который «добавляется». Ссылка на новый массив записывается в переменную массива. Создается иллюзия, что в исходном массиве появился новый элемент.

Рассмотрим представленный ниже программный код.

Листинг 9.13. Использование двумерного индексатора

```
using System;
// Класс с двумерным индексатором:
class MyClass{
   // Закрытое поле, являющееся ссылкой на
   // целочисленный массив:
   private int[] vals;
   // Закрытое поле, являющееся ссылкой на
   // символьный массив:
   private char[] ckey;
   // Закрытое поле, являющееся ссылкой на
   // текстовый массив:
   private string[] skey;
   // Закрытый метод для добавления новых элементов
   // в массивы:
```

```
private void add(char a, string b, int n){
   // Локальная переменная для записи размера
   // новых массивов:
   int size;
   // Переменная для символьного массива:
   char[] s;
   // Переменная для текстового массива:
   string[] t;
   // Переменная для целочисленного массива:
   int[] v;
   // Определение размера новых массивов:
   if(vals==null) size=1;    // Если массива нет
   else size=vals.Length+1;  // Если массив существует
   // Создание символьного массива:
   s=new char[size];
   // Значение последнего элемента созданного массива:
   s[s.Length-1]=a;
   // Создание текстового массива:
   t=new string[size];
   // Значение последнего элемента созданного массива:
   t[t.Length-1]=b;
   // Создание целочисленного массива:
   v=new int[size];
   // Значение последнего элемента созданного массива:
   v[v.Length-1]=n;
   // Заполнение созданных массивов:
   for(int k=0; k<size-1; k++){
      s[k]=ckey[k];
      t[k]=skey[k];
      v[k]=vals[k];
   }
   // Новые значения полей:
```

```
      ckey=s;
      skey=t;
      vals=v;
   }
   // Переопределение метода ToString():
   public override string ToString(){
      // Текстовая переменная:
      string txt="Содержимое объекта:\n";
      // Если массив существует:
      if(vals!=null){
         // Перебор элементов массивов:
         for(int k=0; k<ckey.Length; k++){
            // Добавление текста к текущему значению:
            txt+=ckey[k]+": "+skey[k]+": "+vals[k]+"\n";
         }
      }else{// Если массива нет
         // Добавление текста к текущему значению:
         txt+="Пустой объект\n";
      }
      // Результат метода:
      return txt;
   }
   // Целочисленный индексатор с двумя индексами
   // символьного и текстового типа:
   public int this[char a, string b]{
      // Метод вызывается при считывании значения
      // выражения с проиндексированным объектом:
      get{
         // Если массив существует:
         if(vals!=null){
            // Перебор элементов массива:
            for(int k=0; k<ckey.Length; k++){
```

```
            // Если элемент найден:
            if(a==ckey[k]&&b==skey[k]){
               // Значение выражения:
               return vals[k];
            }
         }
      }
      // Код выполняется, если массив не существует или
      // если элемент не найден:
      int res=0;      // Значение для нового элемента
      add(a, b, res); // Добавление нового элемента
      return res;     // Значение выражения
   }
   // Метод вызывается при присваивании значения
   // выражению с проиндексированным объектом:
   set{
      // Если массив существует:
      if(vals!=null){
         // Перебираются элементы массива:
         for(int k=0; k<ckey.Length; k++){
            // Если элемент найден:
            if(a==ckey[k]&&b==skey[k]){
               // Присваивание значения элементу:
               vals[k]=value;
               // Завершение метода:
               return;
            }
         }
      }
      // Если массива нет или если элемент не найден:
      add(a, b, value); // Добавление нового элемента
   }
```

```
   }
}
// Класс с главным методом:
class UsingTwoDimIndexerDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта:
      MyClass obj=new MyClass();
      // Проверка содержимого объекта:
      Console.WriteLine(obj);
      // Проверка значения элемента:
      Console.WriteLine("Значение элемента: "+obj['A',"Первый"]+"\n");
      // Проверка содержимого объекта:
      Console.WriteLine(obj);
      // Присваивание значения элементу:
      obj['B',"Второй"]=200;
      // Присваивание значения элементу:
      obj['C',"Третий"]=300;
      // Проверка содержимого объекта:
      Console.WriteLine(obj);
      // Проверка значения элемента:
      Console.WriteLine("Значение элемента: "+obj['B',"Первый"]+"\n");
      // Проверка значения элемента:
      Console.WriteLine("Значение элемента: "+obj['B',"Второй"]+"\n");
      // Проверка значения элемента:
      Console.WriteLine("Значение элемента: "+obj['A',"Третий"]+"\n");
      // Проверка содержимого объекта:
      Console.WriteLine(obj);
      // Присваивание значения элементу:
      obj['A',"Первый"]=100;
      // Проверка содержимого объекта:
      Console.WriteLine(obj);
```

```
      // Проверка значения элемента:
      Console.WriteLine("Значение элемента: "+obj['A',"Первый"]+"\n");
   }
}
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 9.13)

```
Содержимое объекта:
Пустой объект

Значение элемента: 0

Содержимое объекта:
A: Первый: 0

Содержимое объекта:
A: Первый: 0
B: Второй: 200
C: Третий: 300

Значение элемента: 0

Значение элемента: 200

Значение элемента: 0

Содержимое объекта:
A: Первый: 0
B: Второй: 200
C: Третий: 300
B: Первый: 0
```

```
A: Третий: 0


Содержимое объекта:
A: Первый: 100
B: Второй: 200
C: Третий: 300
B: Первый: 0
A: Третий: 0


Значение элемента: 100
```

В классе `MyClass`, как отмечалось, есть три закрытых массива `vals` (целочисленный), `ckey` (символьный) и `skey` (текстовый). В классе описан закрытый метод `add()` с тремя аргументами. При вызове метода `add()` в каждый из трех массивов добавляется новый элемент. Значения элементов передаются аргументами методу. При вызове метода проверяется условие `vals==null`, истинное в случае, если поле `vals` не ссылается на массив. Если так, то целочисленная переменная `size` получает значение `1`. Если условие ложно, то целочисленная переменная `size` получает значение `vals.Length+1` (на единицу больше размера массива, на который ссылается поле `vals`). Затем создаются три массива (символьный `s`, текстовый `t` и целочисленный `v`), размер которых определяется значением переменной `size`. Последние элементы массивов получают значения, определяемые аргументами метода (команды `s[s.Length-1]=a`, `t[t.Length-1]=b` и `v[v.Length-1]=n`). Прочие элементы массивов заполняются поэлементным копированием, для чего использован оператор цикла (команды `s[k]=ckey[k]`, `t[k]=skey[k]` и `v[k]=vals[k]` в теле оператора цикла). Наконец, командами `ckey=s`, `skey=t` и `vals=v` значениями полям объекта присваиваются ссылки на новые массивы.

Метод `ToString()` переопределен таким образом, что если поля объекта на массивы не ссылаются, то возвращается текстовая строка с информацией о том, что объект пустой. Если массивы существуют, то возвращается текстовая строка со значениями элементов трех массивов объекта (в одной строке значения элементов с одним и тем же индексом).

В `get`-аксессоре индексатора проверяется условие `vals!=null`. Истинность условия означает, что целочисленный массив существует

(а значит, и два других массива тоже). В этом случае синхронно перебираются все элементы символьного и текстового массивов. Для каждого индекса k проверяется условие a==ckey[k]&&b==skey[k] (первый индекс a совпадает с элементом символьного массива, а второй индекс b совпадает с элементом текстового массива). Если совпадение найдено, значением свойства возвращается vals[k] (элемент целочисленного массива с тем же индексом).

В случае, если массивы не существуют или если совпадение не найдено, вызывается метод add(), которым в массивы добавляется по новому элементу (или создаются массивы, если их не было), и значение (нулевое) элемента, добавленного в целочисленный массив, возвращается результатом выражения.

В set-аксессоре выполняются аналогичные действия, но последствия немного другие. Если при просмотре содержимого массивов совпадение найдено (истинно условие a==ckey[k]&&b==skey[k]), то командой vals[k]=value элементу целочисленного массива присваивается значение. После этого инструкцией return завершается выполнение set-аксессора.

Если совпадение не найдено, то с помощью метода add() в массивы добавляется по одному новому элементу.

В главном методе создается объект obj класса MyClass. Проверка показывает (команда Console.WriteLine(obj)), что объект пустой (его поля не ссылаются на массивы). Поэтому при попытке узнать значение выражения obj['A',"Первый"] получаем 0, а в массивы объекта obj добавляется по новому элементу (понятно, что предварительно массивы создаются).

При выполнении команд obj['B',"Второй"]=200 и obj['C',"Третий"]=300 в массивы добавляется еще по два элемента.

Затем последовательно считываются значения выражений obj['B',"Первый"], obj['B',"Второй"] и obj['A',"Третий"]. Для первого и третьего из этих выражений в объекте obj совпадений индексов не обнаруживается, а для второго выражения совпадение есть. Поэтому для второго выражения возвращается значение 200 из целочисленного массива, а в соответствии с первым и третьим выражениями в массивы объекта добавляются новые элементы (в том числе нулевые значения в целочисленный массив).

НА ЗАМЕТКУ

Напомним, что при индексировании объекта символьный и текстовый массивы просматриваются на предмет «двойного» совпадения: индексы должны одновременно совпасть с соответствующими элементами символьного и текстового массивов.

Командой `obj['A',"Первый"]=100` присваивается новое значение уже существующему элементу целочисленного массива. Проверка это подтверждает.

Многомерные индексаторы

Ну, как говорится, на всякий пожарный случай.

из к/ф «Бриллиантовая рука»

Многомерные индексаторы описываются в полной аналогии с двумерными массивами, но только индексов больше, чем два. Для каждого индекса указывается тип и формальное название. Небольшой пример использования многомерного (с четырьмя индексами) индексатора представлен в листинге 9.14.

Листинг 9.14. Многомерный индексатор

```
using System;
// Класс с многомерным индексатором:
class MyClass{
   // Закрытое целочисленное поле:
   private int code;
   // Конструктор с целочисленным аргументом:
   public MyClass(int n){
      // Значение поля:
      code=n;
   }
   // Переопределение метода ToString():
   public override string ToString(){
```

```
      // Результат метода:
      return "Поле code объекта: "+code;
   }
   // Многомерный индексатор:
   public char this[string a, int i, string b, int j]{
      // Метод вызывается при считывании значения
      // выражения с проиндексированным объектом:
      get{
         // Значение выражения:
         return (char)(a[i]-b[j]+code);
      }
      // Метод вызывается при присваивании значения
      // выражению с проиндексированным объектом:
      set{
         // Значение поля:
         code=value-(a[i]-b[j]);
      }
   }
}
// Класс с главным методом:
class MultDimIndexerDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта:
      MyClass obj=new MyClass('A');
      // Проверка значения поля объекта:
      Console.WriteLine(obj);
      // Текстовые переменные:
      string a="Alpha", b="Bravo";
      // Целочисленные переменные:
      int i=2, j=4;
      // Использование индексатора:
```

```
        Console.WriteLine("obj[\"{0}\",{1},\"{2}\",{3}]={4}",a,i,b,j,obj[a,i,b,j]);
        // Использование индексатора:
        obj[a, i, b, j]='F';
        // Проверка значения поля объекта:
        Console.WriteLine(obj);
        // Использование индексатора:
        Console.WriteLine("obj[\"{0}\",{1},\"{2}\",{3}]={4}",a,i,b,j,obj[a,i,b,j]);
        // Новые значения переменных:
        a="Charlie";
        i=1;
        j=2;
        // Использование индексатора:
        Console.WriteLine("obj[\"{0}\",{1},\"{2}\",{3}]={4}",a,i,b,j,obj[a,i,b,j]);
    }
}
```

Ниже показан результат выполнения программы.

Результат выполнения программы (из листинга 9.14)

```
Поле code объекта: 65
obj["Alpha",2,"Bravo",4]=B
Поле code объекта: 69
obj["Alpha",2,"Bravo",4]=F
obj["Charlie",1,"Bravo",2]=L
```

В этом примере класс `MyClass` содержит:

- закрытое целочисленное поле `code`;
- конструктор с целочисленным аргументом (определяет значение поля);
- переопределенный метод `ToString()` (возвращает результатом текстовую строку со значением поля `code`);
- индексатор символьного типа с четырьмя индексами (текстовый, целочисленный, текстовый и целочисленный).

Нас, понятно, интересует индексатор. В его `get`-аксессоре выполняется команда `return (char)(a[i]-b[j]+code)`. Здесь `a` и `b` — текстовые индексы (первый и третий), а `i` и `j` — целочисленные индексы (второй и четвертый). То есть принцип вычисления значения выражения такой: из текстовых индексов берем по одному символу и вычисляем разность их кодов. К полученному значению прибавляется значение целочисленного поля `code`, полученное значение приводится к символьному типу и возвращается как результат. Символы, которые берутся из текстовых индексов, определяются целочисленными индексами в выражении с проиндексированным объектом.

В `set`-аксессоре командой `code=value-(a[i]-b[j])` полю `code` присваивается новое значение. Оно вычисляется по такому принципу: от фактически присваиваемого значения (параметр `value`) отнимается разность кодов символов, определяемых индексами в выражении с проиндексированным объектом.

ПОДРОБНОСТИ

При выполнении арифметических операций с символьными значениями выполняется автоматическое преобразование символьных значений к целочисленным значениям. Другими словами, в арифметических операциях вместо символов используются их коды.

В главном методе мы создаем объект `obj` класса `MyClass`, причем аргументом передается не целое число, а символьное значение `'A'`. Это допустимо, поскольку есть автоматическое преобразование значений символьного типа в целочисленное значение. В результате поле `code` значением получает код символа `'A'`(проверка показывает, что это число `65`).

Для индексирования объекта `obj` мы используем текстовые переменные `a` и `b` с начальными значениями `"Alpha"` и `"Bravo"`, а также целочисленные переменные `i` и `j` с начальными значениями `2` и `4`. Значение выражения `obj[a, i, b, j]` в этом случае вычисляется так: в тексте `"Alpha"` берем символ с индексом `2` (это буква `'p'`), а из текста `"Bravo"` берем символ с индексом `4` (это буква `'o'`). Разница между кодами символов `'p'` и `'o'` равна `1` (между буквами в алфавите одна позиция), и это значение прибавляется к значению `65` поля `code`, что в итоге дает число `66`, и это код символа `'B'`.

При выполнении команды `obj[a, i, b, j]='F'` новое значение поля `code` объекта `obj` вычисляется следующим образом: из кода символа `'F'`(число `70`) вычитается разность кодов символов `'p'` и `'o'`(разность кодов равна `1`), что дает значение `69`. Это новое значение поля `code`. Понятно, что, если мы теперь проверим значение выражения `obj[a, i, b, j]`, оно окажется равным `'F'`.

Наконец, переменная `a` получает новое значение `"Charlie"` (переменная `b` остается со старым значением `"Bravo"`), значение переменной `i` становится равным `1`, а значение переменной `j` теперь равно `2`. Если вычислить значение выражения `obj[a, i, b, j]`, то получим такое:

- Символ с индексом `1` в тексте `"Charlie"` — это буква `'h'`.
- Символ с индексом `2` в тексте `"Bravo"` — это буква `'a'`.
- Разность кодов символов `'h'` и `'a'` — число `7` (между буквами `'h'` и `'a'` семь позиций).
- К числу `7` прибавляется значение `69` поля `code`, и в результате получается число `76`.
- Коду `76` соответствует буква `'L'`.

Непосредственная проверка подтверждает наши выводы.

Перегрузка индексаторов

Ну, я уверен, что до этого не дойдет!

из к/ф «Бриллиантовая рука»

Индексаторы можно *перегружать*. Перегрузка индексаторов означает, что в классе может быть описано несколько индексаторов, которые должны отличаться количеством и/или типом индексов. Пример класса с перегрузкой индексатора можно найти в программе в листинге 9.15.

Листинг 9.15. Перегрузка индексаторов

```
using System;
// Класс с несколькими индексаторами:
class MyClass{
   // Закрытое целочисленное поле:
```

```
private int[] nums;
// Конструктор класса с одним аргументом:
public MyClass(int size){
   // Создание массива:
   nums=new int[size];
   // Заполнение массива:
   for(int k=0; k<nums.Length; k++){
      // Использование индексатора:
      this[k]=k+1;
   }
}
// Переопределение метода ToString():
public override string ToString(){
   // Текстовая переменная:
   string txt="Содержимое объекта:\n";
   // Формирование текстовой строки:
   for(int k=0; k<nums.Length; k++){
      // Используется индексатор с целочисленным
      // индексом:
      txt+=this[k]+(k==nums.Length-1?"\n":" ");
   }
   // Результат метода:
   return txt;
}
// Индексатор с целочисленным индексом:
public int this[int k]{
   // Аксессор для считывания значения:
   get{
      return nums[k%nums.Length];
   }
   // Аксессор для присваивания значения:
   set{
```

```
            nums[k%nums.Length]=value;
        }
    }
    // Индексатор с символьным индексом:
    public int this[char s]{
        // Аксессор для считывания значения:
        get{
            // Используется индексатор с целочисленным
            // индексом:
            return this[s-'a'];
        }
        // Аксессор для присваивания значения:
        set{
            // Используется индексатор с целочисленным
            // индексом:
            this[s-'a']=value;
        }
    }
    // Индексатор с целочисленным и текстовым индексом:
    public int this[int k, string t]{
        // Аксессор для считывания значения:
        get{
            // Используется индексатор с символьным индексом:
            return this[t[k]];
        }
        // Аксессор для присваивания значения:
        set{
            // Используется индексатор с символьным индексом:
            this[t[k]]=value;
        }
    }
    // Индексатор с текстовым и целочисленным индексом:
```

```
   public int this[string t, int k]{
      // Аксессор для считывания значения:
      get{
         // Использование индексатора с целочисленным и
         // текстовым индексом:
         return this[k, t];
      }
      // Аксессор для присваивания значения:
      set{
         // Использование индексатора с целочисленным и
         // текстовым индексом:
         this[k, t]=value;
      }
   }
}
// Класс с главным методом:
class OverloadingIndexerDemo{
   // Главный метод:
   static void Main(){
      // Целочисленная переменная:
      int n=6;
      // Создание объекта:
      MyClass obj=new MyClass(n);
      // Проверка содержимого объекта:
      Console.WriteLine(obj);
      // Поэлементное отображение содержимого массива:
      for(int k=0; k<n+3; k++){
         // Объект с целочисленным индексом:
         Console.Write(obj[k]+" ");
      }
      Console.WriteLine("\n");
      // Объект с целочисленным индексом:
```

```
      obj[1]=7;
      obj[n+3]=8;
      Console.WriteLine(obj);
      // Объект с символьным индексом:
      obj['a']=9;
      obj['k']=0;
      // Проверка содержимого объекта:
      Console.WriteLine(obj);
      Console.WriteLine("Проверка:");
      // Поэлементное отображение содержимого массива:
      for(char s='a'; s<'a'+n+3; s++){
         // Объект с символьным индексом:
         Console.Write(obj[s]+" ");
      }
      Console.WriteLine("\n");
      // Объект с целочисленным и текстовым индексом:
      obj[4,"alpha"]=0;
      obj["bravo",0]=6;
      // Проверка содержимого массива:
      Console.WriteLine(obj);
      // Текстовая переменная:
      string txt="abc";
      Console.WriteLine("Проверка:");
      // Отображение значений элементов массива:
      for(int k=0; k<txt.Length; k++){
         // Объект с двумя индексами:
         Console.WriteLine(obj[k, txt]+": "+obj[txt, k]);
      }
   }
}
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 9.15)

```
Содержимое объекта:
1 2 3 4 5 6

1 2 3 4 5 6 1 2 3

Содержимое объекта:
1 7 3 8 5 6

Содержимое объекта:
9 7 3 8 0 6

Проверка:
9 7 3 8 0 6 9 7 3

Содержимое объекта:
0 6 3 8 0 6

Проверка:
0 : 0
6 : 6
3 : 3
```

В классе `MyClass` есть закрытый целочисленный массив `nums`. Массив создается и заполняется при вызове конструктора. Для заполнения массива запускается оператор цикла. Там (при заданном индексе `k`) выполняется команда `this[k]=k`, в которой использован индексатор. В этой команде ключевое слово `this` обозначает объект, из которого вызывается метод (в данном случае речь идет об объекте, при создании которого вызывается конструктор).

НА ЗАМЕТКУ

Напомним, что ключевое слово `this` является зарезервированным и обозначает объект, из которого вызывается метод.

Инструкция `this[k]` означает, что индексируется объект, для которого вызван конструктор. Этому выражению присваивается значение `k`. Данная команда выполняется в соответствии с тем, как описан `set`-аксессор индексатора с целочисленным индексом (в классе описано несколько индексаторов). В соответствии с кодом `set`-аксессора выполняется команда `nums[k%nums.Length]=value`, которой значение присваивается элементу массива `nums`. Индекс элемента вычисляется выражением `k%nums.Length`. Результатом является остаток от деления фактического значения индекса `k` на размер массива `nums.Length`. Общий эффект такой, что если положительный индекс `k` не превышает верхнюю допустимую границу (значение `nums.Length-1`), то значение выражения `k%nums.Length` совпадает со значением `k`, а при выходе индекса `k` за верхнюю допустимую границу выполняется циклическая перестановка индекса. Такой же подход использован и в `get`-аксессоре индексатора: результатом возвращается значение элемента `nums[k%nums.Length]`, в котором индекс определяется выражением `k%nums.Length`.

Индексатор с целочисленным индексом мы используем и при переопределении метода `ToString()`: в теле метода при формировании текстовой строки (возвращаемой результатом метода) использовано выражение `this[k]`, смысл которого такой же, как описывалось выше.

ПОДРОБНОСТИ

Значение выражения `k==nums.Length-1?"\n":" "` на основе тернарного оператора `?:` вычисляется так: если значение переменной `k` равно значению выражения `nums.Length-1` (значение индекса последнего элемента массива `nums`), результатом выражения является текст `"\n"`. В противном случае результатом выражения является текст `" "`.

Но в классе `MyClass` описаны и другие индексаторы. Там есть индексатор с символьным индексом. В `get`-аксессоре результатом возвращается выражение `this[s-'a']`. Поскольку значением выражения `s-'a'` является число (разность кодов символов), то выражение `this[s-'a']` вычисляется вызовом `get`-аксессора для индексатора с целочисленным индексом. Аналогично, в `set`-аксессоре для индексатора с символьным индексом при выполнении команды `this[s-'a']=value` вызывается `set`-аксессор для индексатора с целочисленным индексом.

Индексатор с двумя индексами (целое число `k` и текст `t`) в `get`-аксессоре содержит команду `return this[t[k]]`. В выражении `this[t[k]]`

индекс t[k] является символом (символ из текста t с индексом k), поэтому данное выражение вычисляется вызовом get-аксессора для индексатора с символьным индексом. То же справедливо и для set-аксессора, в котором выполняется команда this[t[k]]=value.

Еще одна версия индексатора с двумя индексами отличается от описанной выше порядком индексов: теперь первый индекс t текстовый и второй индекс k целочисленный. В get-аксессоре выполняется команда return this[k, t]. В выражении this[k, t] первый индекс целочисленный, а второй индекс текстовый. Аналогично в set-аксессоре выполняется команда this[k, t]=value, содержащая такое же выражение this[k, t]. В обоих случаях речь идет об использовании индексатора с первым целочисленным индексом и вторым текстовым индексом. В итоге мы создали ситуацию, когда порядок, в котором мы указываем целочисленный и текстовый индекс, не имеет значения.

В главном методе программы создается объект obj класса MyClass. Массив в этом объекте содержит 6 (значение переменной n) элементов, заполненных числами от 1 до 6 включительно. Далее мы запускаем оператор цикла, в котором индексная переменная k «выскакивает» за допустимую верхнюю границу. Но проблемы в этом нет, поскольку при индексировании объекта obj выполняется циклическая перестановка индексов.

При выполнении команд obj[1]=7 и obj[n+3]=8 элементы с индексами 1 и 3 массива nums в объекте obj получают новые значения 7 и 8 соответственно. При выполнении команд obj['a']=9 и obj['k']=0 новые значения (9 и 0) получают элементы с индексами 0 и 4.

ПОДРОБНОСТИ

Если мы используем символ в качестве индекса, то запрос выполняется для элемента с индексом, равным разности кодов этого символа и символа 'a'. Поэтому символ 'a' эквивалентен числовому индексу 0, а символ 'k' эквивалентен числовому индексу 10, что с учетом циклической перестановки для массива из шести элементов дает значение 4.

Когда мы перебираем элементы массива, индексируя символьными значениями объект, то при выходе за допустимую верхнюю границу индексы переставляются циклически — после значения последнего элемента массива отображается значение первого (начального) элемента, затем

второго и так далее. Причина в том, что при обработке выражения с символьным индексом на самом деле вычисляется выражение с целочисленным индексом, а для таких случаев предусмотрена циклическая перестановка индексов.

При выполнении команды `obj[4,"alpha"]=0` нулевое значение присваивается элементу массива `nums` с индексом, который является символом с индексом `4` в тексте `"alpha"`. Это символ `'a'`, который соответствует числовому индексу `0` в массиве. Выполнение команды `obj["bravo",0]=6` означает, что значение `6` присваивается элементу массива `nums` с индексом, который является символом с индексом `0` в тексте `"bravo"`. Это символ `'b'`, который соответствует числовому индексу `1` в массиве.

Также в главном методе есть оператор цикла, в котором с помощью индексной переменной `k` перебираются символы из текстовой переменной `txt` (со значением `"abc"`). В консольном окне отображаются значения выражений вида `obj[k, txt]` и `obj[txt, k]`, которые должны совпадать (и они совпадают), поскольку порядок следования индексов не имеет значения.

> **НА ЗАМЕТКУ**
>
> В рассмотренном примере при определении одних индексаторов мы использовали другие индексаторы. Это не является обязательным условием, но очень часто упрощает жизнь и делает код более универсальным.

Резюме

> Нечестная игра! Ты специально мои ходы плохо думал!
>
> *из к/ф «Кин-дза-дза»*

- Кроме полей и методов, в классе могут быть описаны и другие члены: свойства и индексаторы.
- Свойство представляет собой нечто среднее между полем и методом. По способам использования свойство похоже на поле. В общем случае значение свойства можно прочитать и свойству можно

присвоить значение. Обращение к свойству выполняется так же, как и обращение к полю: имя свойства указывается через точку после имени объекта.

- При считывании значения свойства и при присваивании значения свойству вызываются специальные методы, которые называются аксессорами (`get`-аксессор и `set`-аксессор соответственно). Аксессоры описываются при описании свойства.
- При описании свойства указывается тип свойства и его название (также указывается спецификатор уровня доступа). Затем в фигурных скобках описываются аксессоры. Команды, выполняемые при вызове аксессоров, указываются в фигурных скобках. Перед блоком команд для `get`-аксессора указывается ключевое слово `get`. Перед блоком команд для `set`-аксессора указывается ключевое слово `set`.
- Для свойства может быть описан только один аксессор. Если для свойства описан только `get`-аксессор, то такое свойство доступно для чтения, но недоступно для присваивания значения. Если для свойства описан только `set`-аксессор, то такому свойству можно присвоить значение, но прочитать значение свойства нельзя.
- Свойство не определяет область памяти (то есть наличие у объекта свойства не означает, что для этого свойства выделяется память). Свойство не может быть использовано с идентификаторами `ref` и `out`. Свойство может быть статическим.
- Если в классе описан индексатор, то объект такого класса можно индексировать: после имени объекта в квадратных скобках указывается индекс (или индексы). В общем случае можно считывать значение такого выражения или присваивать значение такому выражению.
- При описании индексатора указывается спецификатор уровня доступа, тип индексатора, ключевое слово `this`, а также в квадратных скобках описываются индексы (указывается тип индекса и формальное название). Если индексов несколько, то их описание разделяется запятыми. В блоке, выделенном фигурными скобками, описывается `get`-аксессор и `set`-аксессор. В аксессорах можно использовать индексы, а в `set`-аксессоре также используют ключевое слово `value`, обозначающее присваиваемое значение. Разрешается описать два аксессора или только один аксессор.
- Индексатор не определяет область памяти. Индексатор не может быть статическим, а выражения, подразумевающие использование

индексатора, не могут использоваться с ключевыми словами `ref` и `out`.

- Индексаторы можно перегружать: в классе может быть описано несколько версий индексатора, которые должны отличаться количеством и/или типом индексов.

Задания для самостоятельной работы

Своими мозгами надо играть.

из к/ф «Кин-дза-дза»

1. Напишите программу, в которой есть класс с символьным свойством. Опишите аксессоры для свойства так, чтобы значение свойства попадало в диапазон символов от `'A'` до `'Z'` включительно.

2. Напишите программу, в которой есть класс с целочисленным массивом и целочисленным свойством. При считывании значения свойства оно последовательно и циклически возвращает значения элементов массива. При присваивании значения свойству изменяется значение того элемента, который в данный момент интерпретируется как значение свойства.

3. Напишите программу, в которой есть класс с целочисленным массивом. Опишите в классе свойство, доступное только для считывания значения. Значением свойства является сумма элементов массива.

4. Напишите программу, в которой есть класс с закрытым неотрицательным целочисленным полем. Также в классе должно быть закрытое текстовое поле, содержащее значением восьмеричный код числа из целочисленного поля. Опишите в классе свойство, доступное только для присваивания значения. При присваивании неотрицательного целочисленного значения свойству соответствующее число записывается в целочисленное поле, а в текстовое поле заносится восьмеричный код числа. Опишите еще одно свойство, доступное только для чтения, которое результатом возвращает текст из текстового поля (восьмеричный код числа).

5. Напишите программу, в которой есть класс со статическим свойством. При считывании значения свойства возвращается нечетное число, каждый раз новое: при первом считывании свойства получаем значение `1`,

затем 3, затем 5 и так далее. При присваивании значения свойству определяется порядковый номер числа в последовательности нечетных чисел, начиная с которого будут возвращаться числа. Например, если присвоить свойству значение 5, то при считывании значения свойства получаем число 9 (пятое по порядку нечетное число), затем число 11, затем 13 и так далее.

6. Напишите программу, в которой есть класс с целочисленным массивом и с индексатором. При считывании значения выражения с проиндексированным объектом результатом возвращается значение элемента массива. При присваивании значения выражению с проиндексированным объектом значение присваивается элементу массива. Необходимо описать индексатор так, чтобы при индексировании объекта первый индекс отличался от нуля. Числовые значения, определяющие диапазон изменения индекса (и, соответственно, размер целочисленного массива) при индексировании объекта, передаются аргументами конструктору класса.

7. Напишите программу с классом, в котором есть неотрицательное целочисленное поле. Опишите для класса индексатор (с одним `get`-аксессором) такой, что при индексировании объекта с целочисленным индексом результатом возвращается значение бита в бинарном представлении числа (значение целочисленного поля).

8. Напишите программу с классом, у которого есть неотрицательное целочисленное поле. В классе нужно описать индексатор с целочисленным индексом и `set`-аксессором. Присваивание значения проиндексированному объекту обрабатывается следующим образом. В фактически присваиваемом значении берется только последняя цифра (остаток от деления числа на 10). Индекс определяет разряд в числовом значении поля, в который записывается цифра. Нулевой разряд соответствует единицам, единичный разряд соответствует десяткам, разряд два соответствует сотням и так далее. Например, если объект проиндексирован числом 1 и присваивается значение, заканчивающееся на 5, то это означает, что в числе, которое является значением поля, в разряд десятков (разряд 1) нужно записать цифру 5.

9. Напишите программу с классом, в котором есть двумерный числовой массив. Опишите два индексатора для класса. Двумерный индексатор с двумя целочисленными индексами позволяет прочитать и изменить значение элемента в двумерном массиве, а индексатор с одним целочисленным индексом возвращает результатом значение наибольшего

элемента в строке двумерного массива. Присваивание значения выражению на основе объекта с одним индексом означает присваивание значения тому элементу в строке, который на данный момент имеет наибольшее значение. Строка определяется индексом, указанным при индексировании объекта. Если в строке несколько элементов с наибольшим значением, то используется первый такой элемент.

10. Напишите программу с классом, в котором есть текстовый массив. Опишите в классе одномерный и двумерный индексаторы. Одномерный индексатор позволяет прочитать элемент текстового массива и присвоить новое значение элементу текстового массива. Двумерный индексатор позволяет прочитать символ в элементе текстового массива (первый индекс определяет элемент в текстовом массиве, а второй индекс определяет символ в тексте). Предусмотрите циклическую перестановку индексов в случае, если они выходят за верхнюю допустимую границу.

Глава 10
НАСЛЕДОВАНИЕ

> А как он с вами разговаривает? Вы, человек, достигший вершин лопдопского дна! В конце концов, вы собираетесь быть принцем?
>
> *из к/ф «Формула любви»*

Данная глава посвящена *наследованию* — одному из фундаментальных механизмов любого объектно ориентированного языка, в том числе и языка C#. Среди вопросов и тем, представляющих для нас интерес, будут такие:

- базовые и производные классы, их описание и использование;
- особенности описания конструктора производного класса;
- влияние уровня доступа члена класса на его наследование;
- замещение полей и методов;
- виртуальные методы и их переопределение;
- возможность использовать объектные переменные базового класса при работе с объектами производного класса.

Как всегда, будет много примеров, которые иллюстрируют способы применения технологий, описываемых в главе.

Знакомство с наследованием

— Где ваша родина?
— Не знаю. Я родился па корабле, но куда он плыл и откуда, никто не помпит.

из к/ф «Формула любви»

Идея наследования достаточно проста: вместо того чтобы создавать новый класс, так сказать, «на пустом месте», мы можем создать его на основе уже существующего класса. Класс, который используется как основа для создания нового класса, называется *базовым*. Класс, который создается на основе базового класса, называется *производным*. Главный (но далеко не единственный) эффект от создания производного класса на основе базового состоит в том, что *открытые* (и *защищенные*) члены из базового класса автоматически добавляются (то есть наследуются) в производный класс.

НА ЗАМЕТКУ

Здесь есть как минимум «экономия» на наборе программного кода: нет необходимости в теле производного класса заново вводить тот код, который наследуется из базового класса. Но имеются и другие положительные эффекты. Допустим, через какое-то время понадобится внести изменения в исходный код, реализованный в базовом классе. Если бы использовалась технология «копирования и вставки» кода, то правки пришлось бы вносить во всех местах, содержащих соответствующий код. Если же используется механизм наследования, то правки достаточно внести лишь в программный код базового класса. Есть и другие преимущества использования механизма наследования. Их мы также будем обсуждать.

ПОДРОБНОСТИ

Выше упоминались защищенные члены класса. Защищенные члены класса описываются с ключевым словом `protected`. Защищенные члены класса, так же как и закрытые (описываются с ключевым словом `private` или без спецификатора доступа), доступны только в пределах программного кода класса. Защищенные члены класса от закрытых отличаются тем, что наследуются в производном классе. Особенности наследования членов базового класса с разным уровнем доступа описываются в этой главе, но немного позже.

Технически наследование реализуется достаточно просто. Базовый класс каким-либо особым образом описывать не нужно. Это самый обычный класс. В описании производного класса после названия класса через двоеточие указывается название базового класса. Так, если мы описываем класс `MyClass` и хотим, чтобы он был производным от уже существующего класса `Base`, то шаблон для описания класса `MyClass` будет выглядеть следующим образом:

```
class MyClass: Base{
   // Код производного класса
}
```

В теле класса `MyClass` описываются только те члены, которые «добавляются» в дополнение к членам, наследуемым из класса `Base`. Повторно описывать члены класса `Base` в классе `MyClass` не нужно. При этом мы можем обращаться к унаследованным членам, хотя формально они в классе `MyClass` не описаны.

НА ЗАМЕТКУ

То, что унаследованные члены базового класса не нужно описывать в производном классе, совсем не означает, что их нельзя там описать. Такое «повторное описание» тоже допустимо. В этом случае имеет место замещение членов и переопределение методов. Данные вопросы мы обсудим немного позже.

Также стоит заметить, что у производного класса может быть только один базовый класс: нельзя создать производный класс на основе сразу нескольких базовых классов. Вместе с тем производный класс сам может быть базовым классом для другого класса. Проще говоря, мы можем реализовать цепочку наследования: на основе базового класса создается производный класс, а на его основе создается еще один производный класс и так далее. Понятно, что один и тот же класс может быть базовым сразу для нескольких производных классов.

Небольшой пример, в котором используется наследование классов, представлен в листинге 10.1.

Листинг 10.1. Знакомство с наследованием

```
using System;
// Базовый класс:
class Base{
```

```
   // Открытое целочисленное поле:
   public int code;
   // Открытый метод:
   public void show(){
      // Отображение значения поля:
      Console.WriteLine("Поле code: "+code);
   }
}
// Производный класс:
class MyClass: Base{
   // Открытое символьное поле:
   public char symb;
   // Открытый метод:
   public void display(){
      // Отображение значения символьного поля:
      Console.WriteLine("Поле symb: "+symb);
      // Вызов унаследованного метода:
      show();
   }
   // Открытое свойство:
   public int number{
      // Аксессор для считывания значения:
      get{
         // Обращение к унаследованному полю:
         return code;
      }
      // Аксессор для присваивания значения:
      set{
         // Обращение к унаследованному полю:
         code=value;
      }
   }
```

```
}
// Класс с главным методом:
class InheritDemo{
   static void Main(){
      // Создание объекта производного класса:
      MyClass obj=new MyClass();
      // Присваивание значений полям:
      obj.code=100;
      obj.symb='A';
      // Вызов метода:
      obj.display();
      // Использование свойства:
      obj.number=200;
      Console.WriteLine("Свойство number: "+obj.number);
      // Вызов метода:
      obj.show();
   }
}
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 10.1)

```
Поле symb: A
Поле code: 100
Свойство number: 200
Поле code: 200
```

Пример очень простой. Мы описали базовый класс `Base`, в котором есть целочисленное поле `code` и метод `show()` без аргументов, не возвращающий результат. При вызове метода в консольном окне отображается значение числового поля объекта, из которого вызывается метод.

На основе класса `Base` путем наследования создается класс `MyClass`. Непосредственно в этом классе описано символьное поле `symb`, метод `display()` и целочисленное свойство `number`. При этом в программном коде класса `MyClass` мы использовали обращение к полю `code`

и методу show(), хотя непосредственно в теле класса они не описывались. Это стало возможным благодаря тому, что поле code и метод show() наследуются из класса Base. Поэтому в классе MyClass они присутствуют, и мы их можем использовать так, как если бы они были описаны в классе MyClass.

Если более детально, то при вызове метода display() сначала отображается значение символьного поля объекта, после чего вызывается метод show(). Метод show(), в соответствии с тем, как он описан в классе Base, отображает в консольном окне значение целочисленного поля объекта, из которого метод вызывается (а вызывается он в данном случае из объекта производного класса).

Свойство number в производном классе непосредственно связано с полем code, которое наследуется из базового класса. При считывании значения свойства number возвращается значение поля code, а при присваивании значения свойству number значение фактически присваивается полю code.

В главном методе программы командой MyClass obj=new MyClass() создается объект obj производного класса MyClass. Как видим, процедура создания объекта осталась той же. Созданный объект obj имеет все поля, методы и прочие члены, которые описаны в классе MyClass или унаследованы в этом классе из базового класса Base. В частности, у объекта obj есть поля code и symb, методы display() и show(), а также свойство number. Командами obj.code=100 и obj.symb='A' полям объекта присваиваются значения. При вызове метода display() (команда obj.display()) в консольном окне отображаются значения полей объекта obj.

Командой obj.number=200 значение присваивается свойству number, а в реальности новое значение получает поле code. При проверке значения свойства number с помощью команды Console.WriteLine("Свойство number: "+obj.number) получаем такое же значение, как и при выполнении команды obj.show(), которой отображается значение поля code.

НА ЗАМЕТКУ

В рассмотренном примере мы описали класс Base, но объекты на основе этого класса не создавались. Причина в том, что в объектах класса Base нет ничего особенного — обычные объекты

обычного класса. Гипотетически, если бы на основе класса `Base` был создан объект, то у такого объекта было бы поле `code` и метод `show()`. И этот объект никак бы не был связан с объектом, созданным на основе класса `MyClass`. Наследование устанавливает связь между классами, а не между объектами.

Наследование и уровни доступа

> А этот пацак все время говорит на языках, продолжения которых не знает!
>
> *из к/ф «Кин-дза-дза»*

Как отмечалось выше, при наследовании из базового класса в производный «перекочевывают» открытые и защищенные члены класса. Пришло время более детально осветить этот вопрос.

Итак, при описании члена класса могут использоваться следующие спецификаторы уровня доступа: `public`, `private`, `protected` и `internal`. Есть еще вариант, что член класса будет описан вообще без спецификатора уровня доступа. Разберем, чем чревата каждая из ситуаций.

- Если член класса описан с ключевым словом `public` (открытый член класса), то он доступен из любого места программного кода. Это наивысший уровень доступности.
- Член класса, который описан с ключевым словом `private` (закрытый член класса), доступен в программном коде в пределах класса, в котором описан. Вне пределов класса напрямую обратиться к такому члену нельзя.
- Если член класса описан с ключевым словом `protected` (защищенный член класса), то он доступен в пределах программного кода своего класса, а также классов, которые являются производными от класса, в котором описан член. Проще говоря, отличие защищенного члена класса от закрытого члена класса состоит в том, что защищенный член класса наследуется, а закрытый — нет. Причем немаловажное значение имеет, что мы вкладываем в понятие «наследуется». Эта ситуация обсуждается далее.
- Если член класса описан без спецификатора уровня доступа, то он является закрытым членом класса.

- Член класса, описанный с ключевым словом `internal` (внутренний член класса), доступен в программном коде в пределах сборки (упрощенно — это совокупность инструкций программы на промежуточном языке, метаданных и ссылок на используемые в программе ресурсы). Если сравнивать в плане доступности `internal`-член с `public`-членом, то последний доступен не только в пределах сборки, но и в других сборках.

ПОДРОБНОСТИ

Член класса может быть описан только с одним спецификатором уровня доступа. Исключение составляет комбинация `protected internal`. Член класса, описанный с комбинацией ключевых слов `protected internal`, доступен в классе и его подклассах, но только в пределах сборки.

По умолчанию классы доступны в пределах сборки. Для того чтобы класс был доступен в других сборках, его описывают с ключевым словом `public`. Такой класс называют открытым.

Если класс описан с ключевым словом `sealed` (указывается перед ключевым словом `class` в описании класса), то такой класс не может быть базовым — то есть на основе `sealed`-класса нельзя создать производный класс. Например, класс `String`, на основе которого реализуются текстовые значения, является `sealed`-классом.

Мы постараемся разобраться с тем, что происходит при наследовании базового класса, в котором есть защищенные и закрытые члены класса. Для этого рассмотрим пример в листинге 10.2.

Листинг 10.2. Наследование и уровни доступа

```
using System;
// Базовый класс с закрытым и защищенным полем:
class Alpha{
   // Закрытое целочисленное поле:
   private int num;
   // Защищенное символьное поле:
   protected char symb;
   // Открытый метод для считывания значения
   // целочисленного поля:
   public int getNum(){
```

```
      return num;
   }
   // Открытый метод для считывания значения
   // символьного поля:
   public char getSymb(){
      return symb;
   }
   // Открытый метод для присваивания значения
   // целочисленному полю:
   public void setNum(int n){
      num=n;
   }
   // Открытый метод для присваивания значения
   // символьному полю:
   public void setSymb(char s){
      symb=s;
   }
}
// Производный класс:
class Bravo: Alpha{
   // Открытое символьное свойство:
   public char symbol{
      // Аксессор для считывания значения свойства:
      get{
         // Обращение к защищенному унаследованному полю:
         return symb;
      }
      // Аксессор для присваивания значения свойству:
      set{
         // Обращение к защищенному унаследованному полю:
         symb=value;
      }
```

```
   }
   // Открытое целочисленное свойство:
   public int number{
      // Аксессор для считывания значения свойства:
      get{
         // Вызов открытого унаследованного метода,
         // возвращающего значение закрытого поля
         // из базового класса:
         return getNum();
      }
      // Аксессор для присваивания значения свойству:
      set{
         // Вызов открытого унаследованного метода для
         // присваивания значения закрытому полю
         // из базового класса:
         setNum(value);
      }
   }
}
// Класс с главным методом:
class PrivateAndProtectedDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта базового класса:
      Alpha A=new Alpha();
      // Вызов метода для присваивания значения
      // целочисленному полю:
      A.setNum(100);
      // Вызов метода для присваивания значения
      // символьному полю:
      A.setSymb('A');
      // Вызов методов для считывания значений полей:
```

```
        Console.WriteLine("Объект A: {0} и {1}", A.getNum(), A.getSymb());
        // Создание объекта производного класса:
        Bravo B=new Bravo();
        // Вызов метода для присваивания значения
        // целочисленному полю:
        B.setNum(200);
        // Вызов метода для присваивания значения
        // символьному полю:
        B.setSymb('B');
        // Вызов методов для считывания значений полей:
        Console.WriteLine("Объект B: {0} и {1}", B.getNum(), B.getSymb());
        // Присваивание значения целочисленному свойству:
        B.number=300;
        // Присваивание значения символьному свойству:
        B.symbol='C';
        // Считывание значений свойств:
        Console.WriteLine("Объект B: {0} и {1}", B.number, B.symbol);
    }
}
```

Ниже представлен результат выполнения программы.

Результат выполнения программы (из листинга 10.2)

```
Объект A: 100 и A
Объект B: 200 и B
Объект B: 300 и C
```

Мы описали класс `Alpha`, в котором есть закрытое целочисленное поле `num`, защищенное символьное поле `symb` и четыре открытых метода. С помощью методов `getNum()` и `getSymb()` можно получить значение соответственно целочисленного и символьного поля, а методы `setNum()` и `setSymb()` позволяют присвоить значения полям.

В главном методе программы командой `Alpha A=new Alpha()` создается объект `A` класса `Alpha`. Для присваивания значений полям этого объекта использованы команды `A.setNum(100)` и `A.setSymb('A')`. Для

считывания значений полей мы используем выражения A.getNum() и A.getSymb(). Другого способа обращения к полям объекта нет, поскольку оба поля (и закрытое поле num, и защищенное поле symb) вне пределов кода класса Alpha недоступны. Мы не можем обратиться через объект A напрямую к этим полям. Нужны «посредники», которыми в данном случае выступают открытые методы setNum(), setSymb(), getNum() и getSymb(). То есть в плане доступности поле symb не отличается от поля num. Разница между полями проявляется при наследовании.

Класс Bravo создается путем наследования класса Alpha. Что же «получает в наследство» класс Bravo? Во-первых, это все открытые методы. Во-вторых, в классе Bravo наследуется поле symb, поскольку оно не закрытое, а защищенное. Причем в классе Bravo это поле остается защищенным. То есть мы можем обратиться к полю symb только в теле класса Bravo. Что касается поля num, то оно не наследуется. Причем «не наследуется» следует понимать в том смысле, что в классе Bravo мы не можем обратиться к полю num. Класс Bravo «ничего не знает» о существовании этого поля, но поле в реальности существует. В теле класса Bravo мы можем обратиться к нему с помощью методов getNum() и setNum(), которые являются открытыми и наследуются из класса Alpha. Именно поэтому в описании целочисленного свойства number в классе Bravo для обращения к полю num использованы методы getNum() и setNum(). В описании символьного свойства symbol мы, по аналогии, могли бы использовать методы getSymb() и setSymb() для получения доступа к полю symb. Но пошли другим путем: поскольку поле symb наследуется, то мы воспользовались возможностью непосредственного обращения к нему.

В главном методе программы мы с помощью команды Bravo B=new Bravo() создаем объект B класса Bravo. Через этот объект мы можем напрямую обращаться к четырем методам и свойствам number и symbol. Поле symb является защищенным, поэтому вне пределов кода класса Bravo оно не доступно. А поле num вообще не наследуется (в указанном выше смысле). Сначала командами B.setNum(200) и B.setSymb('B') полям присваиваются значения, а проверяем мы эти значения с помощью выражений B.getNum() и B.getSymb(). Но есть и другой путь для получения доступа к полям: с помощью свойств. Так, командами B.number=300 и B.symbol='C' записываем в поля новые значения, а с помощью инструкций B.number и B.symbol проверяем результат.

Наследование и конструкторы

> Ален ноби ностра алис, что означает: ежели один человек построил, другой завсегда разобрать может.
>
> *из к/ф «Формула любви»*

Выше мы столкнулись с ситуацией, когда в базовом классе есть закрытое поле, к которому обращается открытый метод. Метод наследуется в производном классе и при вызове из объекта производного класса обращается к полю, которое не наследуется. Это ненаследуемое поле физически наличествует у объекта, но оно недоступно для прямого обращения: мы не можем даже в теле производного класса напрямую обратиться к полю — только через унаследованный метод. Такая ситуация может показаться, на первый взгляд, странной, но в действительности это очень разумный механизм. Понять, как он реализуется, будет легче, если учесть, что при создании объекта производного класса сначала вызывается конструктор базового класса. В рассмотренных выше примерах мы конструкторы явно не описывали ни для базового класса, ни для производного класса. Поэтому использовались конструкторы по умолчанию.

Упрощенная схема создания объекта производного класса в этом случае выглядит так: сначала вызывается конструктор по умолчанию базового класса, и для всех полей объекта, в том числе и закрытых, выделяется место в памяти. Затем в игру вступает собственно конструктор по умолчанию для производного класса, в результате чего место выделяется для тех полей, которые описаны непосредственно в производном классе. Поэтому, даже если поле в базовом классе является закрытым, в объекте производного класса оно все равно будет присутствовать, поскольку место под это поле выделяется при вызове конструктора базового класса.

ПОДРОБНОСТИ

При создании объекта производного класса сначала выполняются команды из конструктора базового класса, а потом выполняются те команды, которые собственно описаны в конструкторе производного класса. При удалении объекта производного класса из памяти сначала выполняются команды из деструктора производного класса, а затем команды из деструктора базового класса. Таким образом, порядок вызова деструкторов обратен к порядку вызова конструкторов.

Теперь представим себе, что в базовом классе несколько конструкторов и среди них нет конструктора без аргументов. Сразу возникает два вопроса: какую версию базового класса использовать при создании объекта производного класса и как передать аргументы этой версии конструктора? Решение проблемы состоит в особом способе описания конструктора производного класса, а именно в описании конструктора производного класса после закрывающей круглой скобки через двоеточие указывается ключевое слово `base`. После него в круглых скобках указывают аргументы, передаваемые конструктору базового класса. Если аргументы конструктору базового класса передавать не нужно, то скобки оставляют пустыми. В теле конструктора производного класса размещают команды, выполняемые после выполнения команд из конструктора базового класса. Шаблон описания конструктора производного класса выглядит так (жирным шрифтом выделены основные элементы шаблона):

```
public Конструктор(аргументы) : base(аргументы) {
   // Команды конструктора производного класса
}
```

В листинге 10.3 представлена программа, в которой в базовом и производном классе описываются конструкторы.

Листинг 10.3. Конструктор производного класса

```
using System;
// Базовый класс:
class Alpha{
   // Целочисленное поле:
   public int code;
   // Конструктор с одним аргументом:
   public Alpha(int n){
      code=n;
      Console.WriteLine("Alpha (один аргумент): {0}", code);
   }
   // Конструктор без аргументов:
   public Alpha(){
      code=123;
      Console.WriteLine("Alpha (без аргументов): {0}", code);
```

```
    }
}
// Производный класс:
class Bravo: Alpha{
    // Символьное поле:
    public char symb;
    // Конструктор с двумя аргументами:
    public Bravo(int n, char s): base(n){
        symb=s;
        Console.WriteLine("Bravo (два аргумента): {0} и \'{1}\'", code, symb);
    }
    // Конструктор с целочисленным аргументом:
    public Bravo(int n): base(n){
        symb='A';
        Console.WriteLine("Bravo (int-аргумент): {0} и \'{1}\'", code, symb);
    }
    // Конструктор с символьным аргументом:
    public Bravo(char s): base(321){
        symb=s;
        Console.WriteLine("Bravo (char-аргумент): {0} и \'{1}\'", code, symb);
    }
    // Конструктор без аргументов:
    public Bravo(): base(){
        symb='O';
        Console.WriteLine("Bravo (без аргументов): {0} и \'{1}\'", code, symb);
    }
}
// Класс с главным методом:
class InheritAndConstrDemo{
    // Главный метод:
    static void Main(){
        // Создание объекта базового класса
```

```
      // (конструктор без аргументов):
      Alpha A1=new Alpha();
      Console.WriteLine();
      // Создание объекта базового класса
      // (конструктор с одним аргументом):
      Alpha A2=new Alpha(100);
      Console.WriteLine();
      // Создание объекта производного класса
      // (конструктор с двумя аргументами):
      Bravo B1=new Bravo(200,'B');
      Console.WriteLine();
      // Создание объекта производного класса
      // (конструктор с целочисленным аргументом):
      Bravo B2=new Bravo(300);
      Console.WriteLine();
      // Создание объекта производного класса
      // (конструктор с символьным аргументом):
      Bravo B3=new Bravo('C');
      Console.WriteLine();
      // Создание объекта производного класса
      // (конструктор без аргументов):
      Bravo B4=new Bravo();
   }
}
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 10.3)

```
Alpha (без аргументов): 123

Alpha (один аргумент): 100

Alpha (один аргумент): 200
```

```
Bravo (два аргумента): 200 и 'B'

Alpha (один аргумент): 300
Bravo (int-аргумент): 300 и 'A'

Alpha (один аргумент): 321
Bravo (char-аргумент): 321 и 'C'

Alpha (без аргументов): 123
Bravo (без аргументов): 123 и '0'
```

В базовом классе `Alpha` есть целочисленное поле `code` и две версии конструктора: с целочисленным аргументом и без аргументов. В каждой версии конструктора полю `code` присваивается значение, после чего в консольное окно выводится сообщение с названием класса `Alpha`, информацией о количестве аргументов конструктора и значением поля `code`. Это сделано для того, чтобы по сообщениям в консольном окне можно было проследить порядок вызова конструкторов. Пример создания объектов базового класса с использованием конструктора без аргументов и с одним аргументом есть в главном методе программы.

В производном классе `Bravo`, кроме наследуемого поля `code`, описано еще и символьное поле `symb`. Версий конструкторов в классе `Bravo` четыре: с двумя аргументами, с целочисленным аргументом, с символьным аргументом и без аргументов. В каждой из этих версий конструктора вызывается одна из версий конструктора базового класса. Например, в версии конструктора с двумя аргументами первый аргумент передается конструктору базового класса, а второй аргумент определяет значение символьного поля. Если конструктору производного класса передается один целочисленный аргумент, то он будет передан конструктору базового класса, а символьное поле получит значение `'A'`. Вызов конструктора производного класса с символьным аргументом приведет к тому, что конструктор базового класса будут вызван с аргументом `321`, а символьный аргумент конструктора производного класса при этом определяет значение символьного поля. Наконец, в конструкторе производного класса без аргументов вызывается конструктор базового класса без аргументов. В теле каждой версии конструктора для производного класса есть команда, которой выводится сообщение с названием класса `Bravo`, количеством аргументов конструктора и значениями полей.

Главный метод программы содержит примеры создания объектов класса `Bravo` с использованием разных версий конструкторов. По сообщениям, которые появляются в консольном окне при создании объектов, можно проследить последовательность вызова конструкторов.

Еще один пример, представленный в листинге 10.4, иллюстрирует последовательность вызова конструкторов и деструкторов при создании и удалении объекта производного класса, когда имеет место цепочка наследования.

НА ЗАМЕТКУ

Имеется в виду *многоуровневое наследование*, при котором производный класс сам является базовым для другого производного класса, а этот производный класс также является базовым для следующего производного класса и так далее.

В представленной далее программе описывается базовый класс `Alpha`, на основе которого создается класс `Bravo`, а на основе этого класса создается класс `Charlie`. Каждый из классов содержит описание конструктора и деструктора. В каждом классе конструктор и деструктор выводят в консольное окно сообщение. В главном методе программы создается анонимный объект класса `Charlie`. Интересующий нас программный код представлен ниже.

Листинг 10.4. Конструкторы и деструкторы при многоуровневом наследовании

```
using System;
// Базовый класс:
class Alpha{
   // Конструктор:
   public Alpha(){
      Console.WriteLine("Конструктор класса Alpha");
   }
   // Деструктор:
   ~Alpha(){
      Console.WriteLine("Деструктор класса Alpha");
   }
```

```
}
// Производный класс от класса Alpha:
class Bravo: Alpha{
   // Конструктор:
   public Bravo(): base(){
      Console.WriteLine("Конструктор класса Bravo");
   }
   // Деструктор:
   ~Bravo(){
      Console.WriteLine("Деструктор класса Bravo");
   }
}
// Производный класс от класса Bravo:
class Charlie: Bravo{
   // Конструктор:
   public Charlie(): base(){
      Console.WriteLine("Конструктор класса Charlie");
   }
   // Деструктор:
   ~Charlie(){
      Console.WriteLine("Деструктор класса Charlie");
   }
}
// Класс с главным методом:
class InheritConstrDestrDemo{
   // Главный метод:
   static void Main(){
      // Создание анонимного объекта:
      new Charlie();
   }
}
```

При выполнении программы получим такой результат.

Результат выполнения программы (из листинга 10.4)

```
Конструктор класса Alpha
Конструктор класса Bravo
Конструктор класса Charlie
Деструктор класса Charlie
Деструктор класса Bravo
Деструктор класса Alpha
```

При выполнении команды `new Charlie()` создается анонимный объект класса `Charlie`. Это самый обычный объект, просто ссылка на него не записывается в объектную переменную (в данном случае в этом просто нет необходимости). При вызове конструктора класса `Charlie` сначала выполняется код конструктора класса `Bravo`, который является базовым для класса `Charlie`. Но выполнение кода конструктора класса `Bravo` начинается с выполнения кода конструктора класса `Alpha`, являющегося базовым для класса `Bravo`. В результате получается, что конструкторы выполняются, начиная с конструктора самого первого базового класса в цепочке наследования.

Сразу после создания анонимного объекта класса `Charlie` программа завершает свое выполнение, и, следовательно, созданный объект удаляется из памяти. Вызывается деструктор класса `Charlie`. После выполнения команд из тела деструктора класса `Charlie` вызывается деструктор класса `Bravo`. Когда выполнены команды из деструктора класса `Bravo`, вызывается деструктор класса `Alpha`. Как отмечалось ранее, порядок вызова конструкторов и деструкторов легко проследить по сообщениям, которые появляются в консольном окне.

НА ЗАМЕТКУ

Напомним, что для запуска приложения на выполнение в среде Visual Studio так, чтобы после выполнения программы консольное окно не закрывалось автоматически, следует нажать комбинацию клавиш <Ctrl>+<F5>.

Объектные переменные базовых классов

— Это Жазель. Француженка. Я признал ее по ноге.
— Нет, это не Жазель. Жазель была брунетка, а эта вся белая.

из к/ф «Формула любви»

Ранее практически каждый раз при работе с объектами мы ссылку на объект записывали в объектную переменную того же класса. Механизм наследования вносит в это строгое правило некоторое разнообразие. Дело в том, что объектная переменная базового класса может ссылаться на объект производного. Допустим, имеется базовый класс `Base`, на основе которого наследованием создается класс `MyClass`. Если мы создадим на основе класса `MyClass` объект, то ссылку на этот объект можно записать не только в объектную переменную класса `MyClass`, но и в объектную переменную базового класса `Base`. Правда, здесь есть одно очень важное ограничение: через объектную переменную базового класса можно получить доступ только к тем членам производного класса, которые объявлены в базовом классе. В нашем случае это означает, что через объектную переменную класса `Base` можно получить доступ только к членам объекта производного класса `MyClass`, объявленным в классе `Base`.

НА ЗАМЕТКУ

Здесь уместно вспомнить про класс `Object` из пространства имен `System` (используют также псевдоним `object` для инструкции `System.Object`), который находится в вершине иерархии наследования всех классов (включая и создаваемые нами). Это обстоятельство позволяет ссылаться объектным переменным класса `Object` на объекты самых разных классов.

Даже несмотря на упомянутое ограничение, данная особенность объектных переменных базовых классов является фундаментально важной и проявляется и используется в самых разных (часто неожиданных) ситуациях. Как небольшую иллюстрацию рассмотрим программу в листинге 10.5.

Листинг 10.5. Объектная переменная базового класса

```
using System;
// Базовый класс:
class Base{
   // Целочисленное поле:
   public int code;
   // Открытый метод:
   public void show(){
      Console.WriteLine("Поле code: "+code);
   }
   // Конструктор с целочисленным аргументом:
   public Base(int n){
      code=n;
   }
   // Конструктор создания копии:
   public Base(Base obj){
      code=obj.code;
   }
}
// Производный класс:
class MyClass: Base{
   // Символьное поле:
   public char symb;
   // Открытый метод:
   public void display(){
      Console.WriteLine("Поле symb: "+symb);
   }
   // Конструктор с двумя аргументами:
   public MyClass(int n, char s): base(n){
      symb=s;
   }
   // Конструктор создания копии:
```

```
   public MyClass(MyClass obj): base(obj){
      symb=obj.symb;
   }
}
// Класс с главным методом:
class BaseObjVarDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта производного класса:
      MyClass A=new MyClass(100,'A');
      // Объектная переменная базового класса:
      Base obj;
      // Объектной переменной базового класса присваивается
      // ссылка на объект производного класса:
      obj=A;
      Console.WriteLine("Используем переменную obj:");
      // Вызов метода через объектную переменную
      // базового класса:
      obj.show();
      // Использовано явное приведение типа:
      ((MyClass)obj).display();
      // Обращение к полю через объектную переменную
      // базового класса:
      obj.code=200;
      // Использовано явное приведение типа:
      ((MyClass)obj).symb='B';
      Console.WriteLine("Используем переменную A:");
      // Вызов методов через объектную переменную
      // производного класса:
      A.show();
      A.display();
      // Создание копии объекта:
```

```
        MyClass B=new MyClass(A);
        // Изменение значений полей исходного объекта:
        A.code=0;
        A.symb='O';
        Console.WriteLine("Используем переменную B:");
        // Проверка значений полей объекта-копии:
        B.show();
        B.display();
    }
}
```

Ниже представлен результат выполнения программы.

Результат выполнения программы (из листинга 10.5)

```
Используем переменную obj:
Поле code: 100
Поле symb: A
Используем переменную A:
Поле code: 200
Поле symb: B
Используем переменную B:
Поле code: 200
Поле symb: B
```

В этой программе мы описываем базовый класс `Base`. У него есть открытое целочисленное поле `code`, метод `show()`, который при вызове отображает в консольном окне значение целочисленного поля, а также в классе есть две версии конструктора. Есть конструктор с одним целочисленным аргументом, и еще имеется конструктор создания копии, аргументом которому передается объект класса `Base`.

ПОДРОБНОСТИ

Под конструктором создания копии обычно подразумевают конструктор, позволяющий создавать объект класса на основе уже существующего объекта того же класса. Обычно (но не обязательно) речь идет о копировании значений полей исходного объекта.

В нашем примере поле `code` создаваемого объекта получает такое же значение, какое есть у поля `code` объекта, переданного аргументом конструктору (команда `code=obj.code` в теле конструктора, а через `obj` обозначен его аргумент). При этом объекты физически разные.

На основе класса `Base` путем наследования создается класс `MyClass`. В этом классе дополнительно описывается символьное поле `symb`, метод `display()`, отображающий в консольном окне значение символьного поля, а также две версии конструктора. В соответствии с описанными версиями объект класса `MyClass` можно создавать, передав два аргумента конструктору (целое число и символ) или передав аргументом уже существующий объект класса `MyClass` (конструктор создания копии). Если при создании объекта класса `MyClass` используются два аргумента, то первый, целочисленный, аргумент передается конструктору базового класса, а второй, символьный, аргумент определяет значение символьного поля. С конструктором создания копии все намного интереснее. Аргумент конструктора `obj` описан как такой, что является объектной ссылкой класса `MyClass`. И эта объектная ссылка передается конструктору базового класса (инструкция `base(obj)` в описании конструктора класса `MyClass`). Здесь имеется в виду версия конструктора базового класса, у которой аргументом является объект. Но дело в том, что в классе `Base` описан конструктор с аргументом, являющимся объектной переменной класса `Base`. А здесь мы по факту передаем аргументом объектную переменную класса `MyClass`. Тем не менее это возможно, и ошибки нет. Почему? Ответ базируется на возможности для объектных переменных базового класса ссылаться на объекты производного класса. Когда вызывается конструктор базового класса, то он «ожидает получить» ссылку на объект класса `Base`. Такая ссылка — аргумент конструктора. Чтобы ее запомнить, создается техническая объектная переменная класса `Base`, в которую будет копироваться значение аргумента.

ПОДРОБНОСТИ

Напомним, что аргументы в методы и конструкторы передаются по значению: на самом деле для переменной, переданной аргументом, создается техническая копия, и все операции выполняются именно с этой копией.

Другими словами, есть специальная «рабочая», автоматически создаваемая переменная, в которую копируется аргумент конструктора. Данная

переменная относится к классу Base, а передается аргументом переменная класса MyClass. Получается, что в объектную переменную класса Base копируется значение переменной класса MyClass. Но это можно делать, поскольку класс Base является базовым для класса MyClass.

При вызове конструктора базового класса с аргументом, являющимся ссылкой на объект производного класса, доступ есть только к тем полям и методам объекта-аргумента, которые объявлены в базовом классе. Но больше и не нужно. Конструктор базового класса выполняет свою работу и присваивает значение полю code создаваемого объекта. После этого командой symb=obj.symb в теле конструктора производного класса значение присваивается символьному полю создаваемого объекта.

В главном методе программы командой MyClass A=new MyClass(100,'A') создается объект производного класса MyClass. Также мы объявляем объектную переменную obj базового класса Base. После выполнения команды obj=A и переменная obj, и переменная A ссылаются на один и тот же объект. Но если через переменную A есть доступ к полям code и symb, а также методам show() и display(), то через переменную obj имеется доступ только к тем полям и методам, которые объявлены в классе Base: это поле code и метод show(). Поэтому команды obj.show() и obj.code=200 являются вполне корректными. А вот если мы хотим через переменную obj получить доступ к полю symb и методу display(), то приходится выполнять явное приведение типа — то есть фактически явно указать, что объект, на который ссылается переменная obj, является объектом класса MyClass. Пример такого подхода дают команды ((MyClass)obj).display() и ((MyClass)obj).symb='B'.

После присваивания с использованием переменной obj новых значений полям code и symb, используя переменную A, убеждаемся, что переменные obj и A действительно ссылаются на один и тот же объект.

Командой MyClass B=new MyClass(A) объект B создается на основе объекта A (так мы проверяем работу конструктора создания копии производного класса). На момент создания объекта B значения его полей такие же, как и у объекта A, но объекты физически разные. Чтобы убедиться в этом, командами A.code=0 и A.symb='O' изменяем значения полей исходного объекта (на основе которого создавалась копия), а с помощью команд B.show() и B.display() убеждаемся, что значения полей объекта B остались такими же, как были у объекта A на момент создания объекта B.

Замещение членов при наследовании

Дядя Вова, ваше пальто идет в моей шапке.

из к/ф «Кин-дза-дза»

В производном классе некоторые поля и методы (и другие члены) наследуются из базового класса, а часть полей и методов описывается непосредственно в производном классе. Может статься, что член, описанный в производном классе, имеет такое же название, как и член, наследуемый из базового класса. Такая ситуация называется *замещением* члена класса. Она вполне законна и может использоваться. Единственное условие состоит в том, что при описании в производном классе поля или метода, название которого совпадает с названием поля или метода, наследуемого из базового класса, необходимо использовать инструкцию `new`.

НА ЗАМЕТКУ

Если не использовать ключевое слово `new`, то программный код скомпилируется, но с предупреждением. Фактически наличие ключевого слова `new` в описании члена класса при замещении свидетельствует о том, что механизм замещения используется осознанно.

Из того, что поле или метод из базового класса замещаются в производном классе, вовсе не следует, что соответствующее поле или метод больше недоступны в производном классе. Замещаемый член класса существует одновременно с тем членом, который описан в производном классе. Просто если мы обращаемся к члену класса по его названию, то по умолчанию обращение выполняется к тому члену, который описан в производном классе. Если нам нужно получить доступ к замещенному члену класса, то перед его названием через точку указывается ключевое слово `base`.

НА ЗАМЕТКУ

Получается, что в производном классе два поля или два метода с одинаковыми названиями. По умолчанию используется то поле или метод, которые описаны непосредственно в производном классе. Чтобы обратиться к члену, унаследованному из базового класса и замещенному из-за описания члена с таким же именем в производном классе, следует использовать ключевое слово `base`.

Пример, в котором используется замещение членов класса, представлен в листинге 10.6.

Листинг 10.6. Замещение членов при наследовании

```
using System;
// Базовый класс:
class Base{
   // Целочисленное поле:
   public int code;
   // Метод для отображения значения поля:
   public void show(){
      Console.WriteLine("Класс Base: "+code);
   }
   // Конструктор с одним аргументом:
   public Base(int n){
      code=n;
   }
}
// Производный класс:
class MyClass: Base{
   // Новое поле замещает одноименное поле,
   // унаследованное из базового класса:
   new public int code;
   // Новый метод замещает одноименный метод,
   // унаследованный из базового класса:
   new public void show(){
      // Вызов версии метода из базового класса:
      base.show();
      // Обращение к полю производного класса:
      Console.WriteLine("Класс MyClass: "+code);
   }
   // Метод для присваивания значения полю, унаследованному
```

```
   // из базового класса и замещенному в производном
   // классе:
   public void set(int n){
      // Обращение к полю, унаследованному из базового
      // класса и замещенному в производном классе:
      base.code=n;
   }
   // Конструктор с двумя аргументами:
   public MyClass(int m, int n): base(m){
      // Присваивание значения полю производного класса:
      code=n;
   }
}
// Класс с главным методом:
class InheritAndHidingDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта производного класса:
      MyClass obj=new MyClass(100,200);
      // Отображение значений полей объекта:
      obj.show();
      Console.WriteLine();
      // Присваивание значения замещенному полю:
      obj.set(300);
      // Присваивание значения замещающему полю:
      obj.code=400;
      // Отображение значений полей объекта:
      obj.show();
   }
}
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 10.6)

```
Класс Base: 100
Класс MyClass: 200

Класс Base: 300
Класс MyClass: 400
```

Базовый класс `Base` содержит целочисленное поле `code`, метод `show()`, отображающий в консоли сообщение с названием класса и значением поля, а также конструктор с одним аргументом, определяющим значение целочисленного поля.

ПОДРОБНОСТИ

Конструктору производного класса `MyClass` передается два целочисленных аргумента. Первый из них становится аргументом конструктора базового класса `Base`. При вызове конструктора базового класса значение получает поле `code`, унаследованное из класса `Base` и замещенное в классе `MyClass`. Второй аргумент конструктора производного класса определяет значение поля `code`, описанного непосредственно в этом классе.

В производном классе `MyClass` также описывается поле `code` и метод `show()`. И поле, и метод описаны с инструкцией `new`. Если мы в теле класса используем идентификатор `code` или вызываем метод `show()`, то имеются в виду именно те члены класса, которые описаны непосредственно в классе `MyClass`. Чтобы получить доступ к полю, унаследованному из базового класса и замещенному в производном, используем инструкцию `base.code`. Чтобы вызвать версию метода `show()` из базового класса, используем инструкцию `base.show()`.

Инструкцию `base.show()` мы используем в теле метода `show()`, который описывается в классе `MyClass`. Это означает, что при вызове метода `show()` из объекта класса `MyClass` первой командой в теле метода вызывается версия метода `show()`, описанная в классе `Base`. В результате в консольном окне отображается значение поля `code`, унаследованного из класса `Base` и замещенного в классе `MyClass`. Затем командой `Console.WriteLine("Класс MyClass: "+code)` отображается значение поля `code`, описанного в классе `MyClass`.

Инструкция `base.code` использована нами в теле метода `set()`, предназначенного для присваивания значения полю `code`, замещенному в производном классе.

В главном методе программы командой `MyClass obj=new MyClass(100,200)` создается объект `obj`, значения полей которого равны `100` (замещенное поле `code`) и `200` (поле `code`, описанное в производном классе). При выполнении команды `obj.show()` из объекта `obj` вызывается версия метода `show()`, описанная в классе `MyClass`. В результате в консольном окне отображаются значения обоих полей.

Командой `obj.set(300)` новое значение присваивается замещенному полю `code`, а командой `obj.code=400` новое значение получает поле `code`, описанное в производном классе. Результат этих операций проверяется с помощью команды `obj.show()`.

Переопределение виртуальных методов

> Ребят, как же это вы без гравицапы пепелац выкатываете из гаража? Это непорядок.
>
> *из к/ф «Кин-дза-дза»*

Выше мы узнали, что в производном классе можно описать метод с таким же названием (и такими же аргументами и типом результата), что и у наследуемого из базового класса метода. В итоге получается, что описанный в производном классе метод «перекрывает», или замещает, метод, унаследованный из базового класса. Но есть еще один механизм, который внешне напоминает замещение метода, но в действительности имеет намного больший потенциал и который достаточно часто используется на практике. Речь идет о *переопределении виртуальных* методов. Сначала мы познакомимся с этим механизмом и научимся его использовать, а затем (в отдельном разделе) сравним переопределение виртуальных методов с замещением методов.

Итак, речь идет о том, что в базовом классе имеется некоторый метод, который наследуется в производном классе. Допустим, что нам нужно, чтобы наследуемый метод в производном классе был, но нас, в силу определенных причин, не устраивает, как этот метод выполняется. Проще говоря, в производном классе мы хотим изменить способ выполнения унаследованного метода. В таком случае метод в производном классе *переопределяют*.

Переопределение метода подразумевает два этапа. Это собственно переопределение, когда в производном классе описывается новая версия метода. Причем такая версия описывается с ключевым словом `override`. Но кроме этого, в базовом классе данный метод должен быть объявлен как *виртуальный*. Для этого в описании метода в базовом классе используют ключевое слово `virtual`.

Пример, иллюстрирующий, как выполняется переопределение виртуальных методов, представлен в листинге 10.7.

Листинг 10.7. Переопределение виртуальных методов

```
using System;
// Базовый класс:
class Alpha{
   // Открытое поле:
   public int alpha;
   // Виртуальный метод:
   public virtual void show(){
      Console.WriteLine("Класс Alpha: {0}", alpha);
   }
   // Конструктор с одним аргументом:
   public Alpha(int a){
      alpha=a;
   }
}
// Производный класс от класса Alpha:
class Bravo: Alpha{
   // Открытое поле:
   public int bravo;
   // Переопределение виртуального метода:
   public override void show(){
      Console.WriteLine("Класс Bravo: {0} и {1}", alpha, bravo);
   }
   // Конструктор с двумя аргументами:
```

```
   public Bravo(int a, int b): base(a){
      bravo=b;
   }
}
// Производный класс от класса Bravo:
class Charlie: Bravo{
   // Открытое поле:
   public int charlie;
   // Переопределение виртуального метода:
   public override void show(){
      Console.WriteLine("Класс Charlie: {0}, {1} и {2}", alpha, bravo, charlie);
   }
   // Конструктор с тремя аргументами:
   public Charlie(int a, int b, int c): base(a, b){
      charlie=c;
   }
}
// Класс с главным методом:
class OverridingDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта класса Alpha:
      Alpha objA=new Alpha(10);
      // Проверка значения поля:
      objA.show();
      Console.WriteLine();
      // Создание объекта класса Bravo:
      Bravo objB=new Bravo(20,30);
      // Объектной переменной базового класса присваивается
      // ссылка на объект производного класса:
      objA=objB;
      // Проверка значений полей:
```

```
        objA.show();
        objB.show();
        Console.WriteLine();
        // Создание объекта класса Charlie:
        Charlie objC=new Charlie(40,50,60);
        // Объектной переменной базового класса присваивается
        // ссылка на объект производного класса:
        objA=objC;
        objB=objC;
        // Проверка значений полей:
        objA.show();
        objB.show();
        objC.show();
    }
}
```

Ниже представлен результат выполнения программы.

Результат выполнения программы (из листинга 10.7)

```
Класс Alpha: 10

Класс Bravo: 20 и 30
Класс Bravo: 20 и 30

Класс Charlie: 40, 50 и 60
Класс Charlie: 40, 50 и 60
Класс Charlie: 40, 50 и 60
```

В этой программе мы описали три однотипных класса: `Alpha`, `Bravo` и `Charlie`. Класс `Alpha` является базовым для класса `Bravo`, а класс `Bravo` является базовым для класса `Charlie`. В классе `Alpha` есть целочисленное поле и конструктор с одним аргументом. В классе `Bravo` есть два целочисленных поля и конструктор с двумя аргументами. В классе `Charlie` есть три целочисленных поля и конструктор с тремя аргументами. Но главный наш интерес связан с методом `show()`,

который описан в базовом классе Alpha и наследуется и переопределяется в классах Bravo и Charlie.

Метод show() описан как виртуальный (с ключевым словом virtual). При вызове метода в консольном окне отображается значение целочисленного поля. Метод переопределяется в классе Bravo. Метод в классе Bravo описан с ключевым словом override. Метод описан так, что при вызове в консольном окне отображается значение двух целочисленных полей. Версия метода в классе Charlie также описана с ключевым словом override, а при вызове метода в консольном окне отображается значение трех целочисленных полей.

В главном методе программы сначала создается объект objA класса Alpha. При вызове метода show() через переменную objA вызывается версия метода из класса Alpha. Далее создается объект objB класса Bravo. Причем переменной objA также присваивается ссылка на этот объект (так можно сделать, поскольку класс Alpha является базовым для класса Bravo). Метод show() вызывается через переменные objA и objB (которые ссылаются на один и тот же объект). В обоих случаях вызывается версия метода show(), описанная в классе Bravo. Наконец, создается объект objC класса Charlie и переменным objA и objB значением присваивается ссылка на этот объект. Через каждую из трех переменных вызывается метод show(). Во всех трех случаях вызывается та версия метода, что описана в классе Charlie.

Таким образом, при вызове виртуального переопределенного метода версия метода определяется не типом переменной, через которую вызывается метод, а классом объекта, на который ссылается объектная переменная. Это очень важная особенность виртуальных методов, и она очень часто используется на практике.

НА ЗАМЕТКУ

Атрибут virtual наследуется. Это означает, что если метод в базовом классе объявлен как виртуальный, то он останется виртуальным при наследовании и переопределении.

Переопределение и замещение методов

А вы опять не переместились, заразы.

из к/ф «Кин-дза-дза»

Теперь пришел черед выяснить, чем замещение методов отличается от переопределения виртуальных методов. Для большей наглядности сразу рассмотрим пример. Он будет несложным, но показательным. Идея, положенная в основу примера, простая. Имеется базовый класс `Alpha`, в котором есть два метода: `hi()` и `hello()`. Метод `hi()` виртуальный, а метод `hello()` обычный (не виртуальный). А еще в классе `Alpha` есть метод `show()`. Это обычный (не виртуальный) метод. При его вызове последовательно вызываются методы `hello()` и `hi()`. Далее, на основе класса `Alpha` создается производный класс `Bravo`. В классе `Bravo` замещается метод `hello()` и переопределяется метод `hi()`. При этом метод `show()` просто наследуется. Таким образом, в классе `Bravo` есть унаследованный метод `show()`, при вызове которого вызываются методы `hello()` и `hi()`. Интрига связана с тем, в каких ситуациях какие версии этих методов будут вызываться. А теперь рассмотрим программный код, представленный в листинге 10.8.

Листинг 10.8. Переопределение и замещение методов

```
using System;
// Базовый класс:
class Alpha{
   // Обычный (не виртуальный) метод:
   public void hello(){
      Console.WriteLine("Метод hello() из класса Alpha");
   }
   // Виртуальный метод:
   public virtual void hi(){
      Console.WriteLine("Метод hi() из класса Alpha");
   }
   // Обычный (не виртуальный) метод:
   public void show(){
```

```
      // Вызов методов:
      hello();
      hi();
      Console.WriteLine();
   }
}
// Производный класс:
class Bravo: Alpha{
   // Замещение метода:
   new public void hello(){
      Console.WriteLine("Метод hello() из класса Bravo");
   }
   // Переопределение виртуального метода:
   public override void hi(){
      Console.WriteLine("Метод hi() из класса Bravo");
   }
}
// Класс с главным методом:
class HidingAndOverridingDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта базового класса:
      Alpha A=new Alpha();
      Console.WriteLine("Выполнение команды A.show():");
      // Вызов метода:
      A.show();
      // Создание объекта производного класса:
      Bravo B=new Bravo();
      Console.WriteLine("Выполнение команды B.hello():");
      // Вызов замещающего метода из производного класса:
      B.hello();
      Console.WriteLine("Выполнение команды B.hi():");
```

```
        // Вызов переопределенного метода:
        B.hi();
        Console.WriteLine("Выполнение команды B.show():");
        // Вызов унаследованного метода:
        B.show();
        Console.WriteLine("После выполнения команды A=B");
        // Объектной переменной базового класса присваивается
        // ссылка на объект производного класса:
        A=B;
        Console.WriteLine("Выполнение команды A.hello():");
        // Вызов замещаемого метода:
        A.hello();
        Console.WriteLine("Выполнение команды A.hi():");
        // Вызов переопределенного метода:
        A.hi();
        Console.WriteLine("Выполнение команды A.show():");
        // Вызов унаследованного метода:
        A.show();
    }
}
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 10.8)

```
Выполнение команды A.show():
Метод hello() из класса Alpha
Метод hi() из класса Alpha

Выполнение команды B.hello():
Метод hello() из класса Bravo
Выполнение команды B.hi():
Метод hi() из класса Bravo
Выполнение команды B.show():
```

```
Метод hello() из класса Alpha
Метод hi() из класса Bravo


После выполнения команды A=B
Выполнение команды A.hello():
Метод hello() из класса Alpha
Выполнение команды A.hi():
Метод hi() из класса Bravo
Выполнение команды A.show():
Метод hello() из класса Alpha
Метод hi() из класса Bravo
```

В главном методе сначала создается объект класса `Alpha`, и ссылка на него записывается в объектную переменную `A` того же класса. Результат вызова метода `show()` через эту переменную вполне ожидаем: вызываются те версии методов `hello()` и `hi()`, которые описаны в классе `Alpha`.

На следующем этапе создается объект класса `Bravo` и ссылка на объект записывается в объектную переменную `B` класса `Bravo`. При выполнении команд `B.hello()` и `B.hi()` убеждаемся, что вызываются версии методов `hello()` и `hi()`, описанные непосредственно в классе `Bravo`. Сюрприз получаем при выполнении команды `B.show()`. Оказывается, что в этом случае для метода `hi()` вызывается версия из класса `Bravo`, а для метода `hello()` вызывается версия из класса `Alpha`.

НА ЗАМЕТКУ

Метод `show()` вызывается из объекта класса `Bravo`, но сам метод описан в классе `Alpha`. Версия виртуального переопределенного метода `hi()` определяется в соответствии с объектом, из которого вызывается метод `show()`. Версия замещаемого в производном классе метода `hello()` определяется по классу, в котором описан вызывающийся метод `show()`.

После выполнения команды `A=B` обе переменные `A` и `B` ссылаются на один и тот же объект класса `Bravo`. В частности, объектная переменная `A` базового класса `Alpha` ссылается на объект производного класса `Bravo`. Через эту переменную мы последовательно вызываем все три

метода (команды `A.hello()`, `A.hi()` и `A.show()`). Вывод очевидный: версия виртуального переопределенного метода определяется по объекту, из которого вызывается метод, а версия замещаемого метода определяется типом объектной переменной.

ПОДРОБНОСТИ

Переопределяется только виртуальный метод. Механизм замещения формально можно применять и к виртуальным методам. Но это все равно будет замещение, а не переопределение метода. Версия метода определяется по объекту, из которого вызывается метод, если метод переопределен.

Переопределение и перегрузка методов

> Вся операция займет не более пятнадцати минут. Пускай отмокает, а я пока соберу барахлишко.
>
> *из к/ф «Бриллиантовая рука»*

Мы познакомились с *переопределением* метода, когда в производном классе описывается новая версия унаследованного метода. В данном случае речь идет о полном совпадении типа, названия и списка аргументов метода в базовом и производном классах. Но еще есть такой механизм, как *перегрузка* методов. При перегрузке метода описывается несколько версий этого метода. Для каждой версии название одно и то же, но вот списки аргументов должны отличаться. Эти два механизма (перегрузка и переопределение) могут использоваться одновременно. В листинге 10.9 представлена программа, в которой на фоне наследования используется перегрузка и переопределение методов.

Листинг 10.9. Перегрузка и переопределение методов

```
using System;
// Базовый класс:
class Alpha{
   // Целочисленное поле:
   public int alpha;
```

```
   // Метод без аргументов:
   public void set(){
      // Значение поля:
      alpha=10;
      // Отображение значения поля:
      Console.WriteLine("Alpha (без аргументов): {0}", alpha);
   }
   // Метод (виртуальный) с одним аргументом:
   public virtual void set(int n){
      // Значение поля:
      alpha=n;
      // Отображение значения поля:
      Console.WriteLine("Alpha (один аргумент):  {0}", alpha);
   }
}
// Производный класс:
class Bravo: Alpha{
   // Целочисленное поле:
   public int bravo;
   // Переопределение виртуального метода:
   public override void set(int n){
      // Присваивание значений полям:
      alpha=n;
      bravo=alpha;
      // Отображение значений полей:
      Console.WriteLine("Bravo (один аргумент):  {0} и {1}", alpha, bravo);
   }
   // Метод с двумя аргументами:
   public void set(int m, int n){
      // Присваивание значений полям:
      alpha=m;
      bravo=n;
```

```
      // Отображение значений полей:
      Console.WriteLine("Bravo (два аргумента):  {0} и {1}", alpha, bravo);
   }
}
// Класс с главным методом:
class OverloadAndOverrideDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта базового класса:
      Alpha A=new Alpha();
      // Вызов методов:
      A.set();
      A.set(20);
      Console.WriteLine();
      // Создание объекта производного класса:
      Bravo B=new Bravo();
      // Вызов методов:
      B.set();
      B.set(30);
      B.set(40,50);
   }
}
```

Ниже показано, как выглядит результат выполнения программы.

Результат выполнения программы (из листинга 10.9)

```
Alpha (без аргументов): 10
Alpha (один аргумент):  20

Alpha (без аргументов): 10
Bravo (один аргумент):  30 и 30
Bravo (два аргумента):  40 и 50
```

У нас есть два класса: `Alpha` и `Bravo`, причем класс `Bravo` является производным от класса `Alpha`. В классе `Alpha` имеется открытое целочисленное поле `alpha`, которое наследуется в классе `Bravo`. Еще в классе `Bravo` появляется целочисленное поле `bravo`. В классе `Alpha` описаны две версии метода `set()`: без аргументов и с одним аргументом. Версия метода `set()` с одним аргументом описана как виртуальная (использовано ключевое слово `virtual`). В классе `Bravo` появляется еще одна версия метода `set()` с двумя аргументами, а версия метода с одним аргументом переопределяется (она описана с ключевым словом `override`). Каждая версия метода описана так, что полю или полям присваиваются значения, после чего в консольное окно выводится сообщение с указанием названия класса, количества аргументов у метода и фактического значения поля или полей объекта. Таким образом, по сообщению, отображаемому в консольном окне при вызове метода, можно однозначно определить, какая версия метода была вызвана.

В главном методе программы мы создаем объект `A` класса `Alpha` и объект `B` класса `Bravo`. Из каждого объекта вызываются разные версии метода `set()`. Для объекта `A` это две версии (обе описаны в классе `Alpha`): без аргументов и с одним аргументом. Для объекта `B` таких версий три. Это, во-первых, версия метода без аргументов, унаследованная из класса `Alpha`. Во-вторых, переопределенная в классе `Bravo` версия метода с одним аргументом. И в-третьих, версия с двумя аргументами, описанная в классе `Bravo`.

Наследование свойств и индексаторов

Бедняга. Ребята, на его месте должен был быть я!

из к/ф «Бриллиантовая рука»

Ранее при изучении вопросов, связанных с наследованием, мы в основном имели дело с полями и методами. Но наследоваться могут также индексаторы и свойства. Более того, они даже могут быть виртуальными (то есть их можно переопределять как виртуальные методы). Ничего удивительного здесь нет, особенно если вспомнить, что механизм реализации свойств и индексаторов базируется на использовании методов-аксессоров. Скромный пример, в котором имеет место наследование индексаторов и свойств, представлен в программе в листинге 10.10.

Листинг 10.10. Наследование индексаторов и свойств

```
using System;
// Базовый класс:
class Alpha{
   // Защищенное целочисленное поле:
   protected int alpha;
   // Закрытый массив:
   private char[] symbs;
   // Конструктор с двумя аргументами:
   public Alpha(int a, string txt){
      // Присваивание значения полю:
      alpha=a;
      // Создание массива на основе текстового аргумента:
      symbs=txt.ToCharArray();
   }
   // Виртуальное свойство:
   public virtual int number{
      // Аксессор для считывания значения:
      get{
         return alpha;
      }
      // Аксессор для присваивания значения:
      set{
         alpha=value;
      }
   }
   // Открытое свойство:
   public int length{
      // Аксессор для считывания значения:
      get{
         return symbs.Length;
      }
```

```
}
// Символьный индексатор с целочисленным индексом:
public char this[int k]{
   // Аксессор для считывания значения:
   get{
      return symbs[k];
   }
   // Аксессор для присваивания значения:
   set{
      symbs[k]=value;
   }
}
// Виртуальный целочисленный индексатор
// с символьным индексом:
public virtual int this[char s]{
   // Аксессор для считывания значения:
   get{
      // Используется индексирование объекта:
      return this[s-'a'];
   }
   // Аксессор для присваивания значения:
   set{
      // Используется индексирование объекта:
      this[s-'a']=(char)value;
   }
}
// Переопределение метода ToString():
public override string ToString(){
   // Текстовая переменная:
   string txt="|";
   // Формируется значение текстовой переменной:
   for(int k=0; k<this.length; k++){
```

```
            txt+=this[k]+"|";
        }
        // Результат метода:
        return txt;
    }
}
// Производный класс:
class Bravo: Alpha{
    // Защищенное целочисленное поле:
    protected int bravo;
    // Конструктор с тремя аргументами:
    public Bravo(int a, int b, string txt): base(a, txt){
        // Значение целочисленного поля:
        bravo=b;
    }
    // Переопределение свойства:
    public override int number{
        // Аксессор для считывания значения:
        get{
            return alpha+bravo;
        }
    }
    // Переопределение индексатора с символьным индексом:
    public override int this[char s]{
        // Аксессор для считывания значения:
        get{
            if(s=='a') return alpha;
            else return bravo;
        }
        // Аксессор для присваивания значения:
        set{
            if(s=='a') alpha=value;
```

```
         else bravo=value;
      }
   }
}
// Класс с главным методом:
class InheritPropIndexerDemo{
   // Главный метод:
   static void Main(){
      // Целочисленная переменная:
      int k;
      // Символьная переменная:
      char s;
      // Создание объекта базового класса:
      Alpha A=new Alpha(100,"ABCD");
      Console.WriteLine("Объект A:");
      // Содержимое символьного массива объекта:
      Console.WriteLine(A);
      // Значение свойства number объекта:
      Console.WriteLine("A.number="+A.number);
      // Присваивание значения свойству number:
      A.number=150;
      // Значение свойства number объекта:
      Console.WriteLine("A.number="+A.number);
      // Индексирование объекта:
      for(k=0, s='a'; k<A.length; k++, s++){
         Console.WriteLine("Символ \'{0}\' с кодом {1}", A[k], A[s]);
      }
      A[0]='E';
      A['b']='A'+10;
      // Содержимое символьного массива объекта:
      Console.WriteLine(A);
      // Создание объекта производного класса:
```

```
      Bravo B=new Bravo(200,300,"EFGHI");
      Console.WriteLine("Объект B:");
      // Содержимое массива объекта:
      Console.WriteLine(B);
      // Значение свойства number объекта:
      Console.WriteLine("B.number="+B.number);
      // Присваивание значения свойству number объекта:
      B.number=400;
      // Значение свойства number объекта:
      Console.WriteLine("B.number="+B.number);
      // Индексирование объекта:
      B['a']=10;
      B['d']=20;
      Console.WriteLine("B[\'a\']="+B['a']);
      Console.WriteLine("B[\'b\']="+B['b']);
      // Проверка значения свойства number объекта:
      Console.WriteLine("B.number="+B.number);
      // Индексирование объекта:
      for(k=0; k<B.length; k++){
         Console.Write(B[k]+" ");
         B[k]=(char)('a'+k);
      }
      Console.WriteLine();
      // Проверка содержимого массива объекта:
      Console.WriteLine(B);
   }
}
```

Как выглядит результат выполнения программы, показано ниже.

Результат выполнения программы (из листинга 10.10)

```
Объект A:
|A|B|C|D|
```

```
A.number=100
A.number=150
Символ 'A' с кодом 65
Символ 'B' с кодом 66
Символ 'C' с кодом 67
Символ 'D' с кодом 68
|E|K|C|D|
Объект B:
|E|F|G|H|I|
B.number=500
B.number=700
B['a']=10
B['b']=20
B.number=30
E F G H I
|a|b|c|d|e|
```

В этой программе мы описываем базовый класс `Alpha` и производный от него класс `Bravo`. В классе `Alpha` есть защищенное целочисленное поле `alpha`, а также закрытый символьный массив `symbs` (поле, являющееся ссылкой на массив). Конструктор класса имеет два аргумента. Первый целочисленный аргумент определяет значение целочисленного поля `alpha`. Второй текстовый аргумент служит основанием для создания символьного массива. Чтобы на основе текста получить массив (состоящий из букв, формирующих текст), мы использовали библиотечный метод `ToCharArray()`.

В классе `Alpha` описано виртуальное (в описании использовано ключевое слово `virtual`) целочисленное свойство `number` (это свойство переопределяется в производном классе). Значением свойства возвращается значение целочисленного поля `alpha`, а при присваивании значения свойству значение присваивается этому полю. Еще одно свойство `length` доступно только для чтения и результатом возвращает размер символьного массива.

В классе `Alpha` описано два индексатора. Есть там символьный индексатор с целочисленным индексом. Результатом выражения

с проиндексированным объектом возвращается значение элемента символьного массива symbs, при присваивании значения проиндексированному объекту оно присваивается элементу массива symbs с соответствующим индексом. Еще есть целочисленный индексатор с символьным индексом. Этот индексатор виртуальный (описан с ключевым словом virtual) и переопределяется в производном классе. В базовом классе он описан так, что для заданного символьного индекса s результатом get-аксессора возвращается значение выражения this[s-'a']. Здесь ключевое слово this обозначает индексируемый объект. Значением выражения s-'a' является целое число, равное разности кодов символов: от кода символьного значения переменной s вычитается код символа 'a'. В итоге получается ключевое слово this, проиндексированное целочисленным индексом. Такое выражение обрабатывается индексатором с целочисленным индексом. Результатом является символ из символьного массива. Но поскольку все это описано в get-аксессоре, который должен возвращать результатом целое число, то символ автоматически будет преобразован в числовое значение (код символа). Таким образом, при индексировании объекта символьным индексом результатом будет возвращаться код символа из символьного массива. Индекс элемента в массиве определяется символьным индексом объекта: символ 'a' соответствует индексу 0, символ 'b' соответствует индексу 1 и так далее.

В set-аксессоре индексатора выполняется команда this[s-'a']=(char)value, которой присваиваемое числовое значение преобразуется в символьное значение, и это символьное значение присваивается элементу символьного массива. Принцип индексации такой же, как и в get-аксессоре.

В классе Alpha переопределен метод ToString(), который результатом возвращает текстовую строку, содержащую символы из символьного массива. При этом в описании метода для определения размера массива мы использовали выражение this.length на основе свойства length с явным указанием ссылки на объект this, а для получения значения элемента массива использовалась инструкция this[k], означающая индексирование объекта.

В главном методе программы объект A класса Alpha создается командой Alpha A=new Alpha(100,"ABCD"). Содержимое символьного массива, формируемого на основе текстовой строки "ABCD", проверяем командой Console.WriteLine(A). В последнем случае для

преобразования объекта A к текстовому формату будет задействован метод ToSring(). При считывании значения свойства number (инструкция A.number) результатом является значение поля alpha. При присваивании значения свойству number (команда A.number=150) значение присваивается полю alpha.

Для отображения символов (из массива) и их кодов используется оператор цикла, в котором синхронно изменяются целочисленная k и символьная s индексные переменные. Выражение вида A[k] результатом дает символ, а выражение вида A[s] возвращает результатом код этого символа.

При выполнении команды A[0]='E' элемент с индексом 0 в символьном массиве symbs получает значение 'E', а при выполнении команды A['b']='A'+10 элемент с индексом 1 в символьном массиве symbs получает значением символ, код которого вычисляется выражением 'A'+10 (число 75, являющееся кодом символа 'K').

Производный класс Bravo создается наследованием класса Alpha. Класс Bravo получает такое «наследство»: защищенное поле alpha, свойство length, индексатор с целочисленным индексом, метод ToString() — все это, так сказать, «в неизменном виде». Также в классе Bravo наследуются и переопределяются: виртуальное свойство number и виртуальный индексатор с символьным аргументом. Кроме этого, в классе Bravo описано защищенное целочисленное поле bravo. У конструктора класса три аргумента (два целочисленных и один текстовый).

При переопределении свойства number мы его еще раз описываем в производном классе, причем используем ключевое слово override. Более того, в производном классе данное свойство описано только с get-аксессором. Если в классе Alpha значение свойства определялось значением поля alpha, то в классе Bravo значение свойства number вычисляется как сумма значений полей alpha и bravo. Поскольку set-аксессор не описан, то будет использоваться set-аксессор свойства number из класса Alpha. Таким образом, если присвоить значение свойству number объекта класса Bravo, то значение будет присвоено полю alpha. Если прочитать значение свойства number объекта класса Bravo, то результатом получим сумму значений полей alpha и bravo.

Индексатор с символьным индексом в классе Bravo определяется совершенно иначе по сравнению с классом Alpha. Теперь если проиндексировать объект символом 'a', то получаем доступ к полю alpha

объекта. Если указать любой другой символьный индекс, получим доступ к полю bravo объекта.

Объект B класса Bravo создается в главном методе программы командой Bravo B=new Bravo(200,300,"EFGHI"). Для проверки содержимого символьного массива объекта используем команду Console.WriteLine(B), при выполнении которой для преобразования объекта B к текстовому формату вызывается метод ToString(), унаследованный из класса Alpha.

ПОДРОБНОСТИ

Символьный массив symbs из класса Alpha в классе Bravo не наследуется в том смысле, что в теле класса Bravo мы не можем напрямую обратиться к массиву по имени. Тем не менее массив существует и доступ к нему осуществляется индексированием объектов, а размер массива можно узнать с помощью свойства length.

При считывании значения свойства number объекта B (инструкция B.number) значением возвращается сумма полей alpha и bravo объекта B. При выполнении команды B.number=400 полю alpha присваивается значение 400.

Командой B['a']=10 полю alpha присваивается значение 10, а при выполнении команды B['d']=20 значение 20 присваивается полю bravo. Соответственно, значением выражения B['a'] возвращается значение поля alpha, а значением выражения B['b'] возвращается значение поля bravo.

ПОДРОБНОСТИ

Если при индексировании объекта B мы указали символьный индекс 'a', то выполняется обращение к полю alpha объекта B. Если мы указали любой другой символьный индекс (например, 'd' или 'b'), то обращение выполняется к полю bravo объекта B.

В операторе цикла индексная переменная k пробегает значения индексов элементов символьного массива в объекте B. Размер массива определяем выражением B.length. За каждый цикл отображается значение элемента массива (инструкция B[k]), после чего командой B[k]=(char)('a'+k) элементу присваивается новое значение. По завершении оператора цикла командой Console.WriteLine(B) проверяется содержимое массива объекта B.

Резюме

Спокойпо, Скрипач, не раздражай даму.

из к/ф «Кин-дза-дза»

- Классы можно создавать на основе уже существующих классов путем наследования. Класс, на основе которого создается новый класс, называется базовым. Тот класс, что создается на основе базового, называется производным классом.
- Для создания производного класса на основе базового в описании производного класса после его имени через двоеточие указывается имя базового класса. Все открытые и защищенные члены базового класса наследуются в производном классе. Закрытые члены базового класса не наследуются в производном классе, в том смысле, что в теле производного класса к ним нельзя обратиться напрямую по имени.
- При создании объекта производного класса сначала вызывается конструктор базового класса, а уже затем выполняются команды, описанные в конструкторе производного класса. В описании конструктора производного класса указывается инструкция вызова конструктора базового класса: это ключевое слово `base` с круглыми скобками, в которых могут передаваться аргументы конструктору базового класса. Инструкция на основе ключевого слова `base` указывается через двоеточие в описании конструктора производного класса после закрывающей круглой скобки.
- При удалении объекта производного класса из памяти сначала выполняются команды в теле деструктора производного класса, а затем вызывается деструктор базового класса.
- Объектная переменная базового класса может ссылаться на объект производного класса. Через объектную переменную базового класса можно получить доступ только к тем членам объекта производного класса, которые объявлены в базовом классе.
- Унаследованные из базового класса члены могут замещаться в производном классе. Для этого соответствующий член описывается в производном классе с ключевым словом `new`. Для обращения к такому члену используется его имя. Для обращения к одноименному члену из базового класса, замещенному в производном классе, перед именем этого члена через точку указывают ключевое слово `base`.

- Если в базовом классе метод объявлен как виртуальный (в описании метода использовано ключевое слово `virtual`), то в производном классе такой метод можно переопределить. Для этого в производном классе описывается новая версия метода. При этом используется ключевое слово `override`.
- Принципиальная разница между замещением и переопределением метода состоит в том, что при замещении метода версия метода для вызова определяется по объектной переменной, через которую вызывается метод, а при переопределении метода версия метода для вызова определяется по объекту, из которого вызывается метод. Наряду с замещением и переопределением методов можно также использовать и перегрузку методов.
- Свойства и индексаторы могут наследоваться. В описании свойств и индексаторов в базовом классе допускается использовать ключевое слово `virtual`, а в производном классе такие свойства и индексаторы могут переопределяться.

Задания для самостоятельной работы

> Если есть на этом Плюке гравицапа, так достанем. Не такое доставали.
>
> *из к/ф «Кин-дза-дза»*

1. Напишите программу, в которой есть базовый класс с защищенным текстовым полем, конструктором с текстовым аргументом и где переопределен метод `ToString()`. На основе базового класса путем наследования создается производный класс. У него появляется еще одно защищенное текстовое поле. Также производный класс должен иметь версии конструктора с одним и двумя текстовыми аргументами, а еще в нем должен быть переопределен метод `ToString()`. В обоих классах метод `ToString()` переопределяется так, что он возвращает строку с названием класса и значение текстового поля или текстовых полей.

2. Напишите программу, в которой есть базовый класс с защищенным текстовым полем. В базовом классе должен быть метод для присваивания значения полю: без аргументов и с одним текстовым аргументом. Объект базового класса создается передачей одного текстового аргумента конструктору. Доступное только для чтения свойство результатом

возвращает длину текстовой строки. Доступный только для чтения индексатор возвращает значением символ из текстовой строки. На основе базового класса создается производный класс. В производном классе появляется дополнительное открытое целочисленное поле. В классе должны быть такие версии метода для присваивания значений полям (используется переопределение и перегрузка метода из базового класса): без аргументов, с текстовым аргументом, с целочисленным аргументом, с текстовым и целочисленным аргументом. У конструктора производного класса два аргумента (целочисленный и текстовый).

3. Напишите программу, в которой есть базовый класс с открытым полем, являющимся ссылкой на целочисленный массив. Конструктору класса при создании передается ссылка на массив, в результате чего создается копия этого массива и ссылка на него записывается в поле объекта. Метод `ToString()` переопределен так, что возвращает текстовую строку со значениями элементов массива. На основе базового класса создается производный класс. В производном классе появляется еще одно открытое поле, являющееся ссылкой на символьный массив. Конструктору производного класса передаются две ссылки: на целочисленный массив и на символьный массив. В результате должны создаваться копии этих массивов, а ссылки на созданные массивы записываются в поля объекта. Метод `ToString()` должен возвращать текстовую строку с содержимым обоих массивов.

4. Напишите программу, в которой на основе базового класса создается производный класс, а на основе этого производного класса создается еще один производный класс (цепочка наследования из трех классов). В первом базовом классе есть открытое целочисленное поле, метод с одним аргументом для присваивания значения полю и конструктор с одним аргументом. Во втором классе появляется открытое символьное поле, метод с двумя аргументами для присваивания значения полям (перегрузка метода из базового класса) и конструктор с двумя аргументами. В третьем классе появляется открытое текстовое поле, метод с тремя аргументами для присваивания значений полям (перегрузка метода из базового класса) и конструктор с тремя аргументами. Для каждого класса определите метод `ToString()` так, чтобы он возвращал строку с названием класса и значениями всех полей объекта.

5. Напишите программу, в которой использована цепочка наследования из трех классов. В первом классе есть открытое символьное поле. Во втором классе появляется открытое текстовое поле. В третьем классе

появляется открытое целочисленное поле. В каждом из классов должен быть конструктор, позволяющий создавать объект на основе значений полей, переданных аргументами конструктору, а также конструктор создания копии.

6. Напишите программу, в которой есть базовый класс с защищенным текстовым полем, конструктор с текстовым аргументом и метод, при вызове которого в консольном окне отображается название класса и значение поля. На основе базового класса создаются два производных класса (оба на основе одного и того же базового). В одном из классов появляется защищенное целочисленное поле, там есть конструктор с двумя аргументами и переопределен метод для отображения значений полей объекта и названия класса. Во втором производном классе появляется защищенное символьное поле, конструктор с двумя аргументами и переопределен метод, отображающий в консоли название класса и значения полей. В главном методе создайте объекты каждого из классов. Проверьте работу метода, отображающего значения полей объектов, для каждого из объектов. Вызовите этот же метод через объектную переменную базового класса, которая ссылается на объект производного класса.

7. Напишите программу, в которой есть базовый класс с открытым текстовым полем. На его основе создается производный класс с дополнительным открытым символьным полем. Опишите в базовом классе виртуальный метод, который при вызове создает и возвращает результатом объект производного класса. Переопределите в производном классе этот метод так, чтобы он создавал и возвращал копию объекта, из которого вызывается.

8. Напишите программу, в которой есть базовый класс с открытым целочисленным полем. В классе описан виртуальный индексатор, позволяющий считывать цифры в десятичном представлении числа (значение поля). На основе базового класса создается производный класс, в котором появляется еще одно открытое целочисленное поле. В производном классе описывается версия индексатора с двумя индексами: первый индекс определяет поле, значение которого используется, а второй индекс определяет разряд, для которого считывается цифра. Индексатор с одним индексом переопределяется так, что вычисления (считывание цифры в десятичном представлении числа) производятся на основе значения, равного сумме значений полей индексируемого объекта.

9. Напишите программу, в которой есть базовый класс с защищенным текстовым полем. В классе имеется виртуальное текстовое свойство,

возвращающее значением текст из текстового поля. При присваивании значения свойству значение присваивается текстовому полю. В классе переопределен метод `ToString()`: он возвращает текстовую строку с названием класса и значением текстового поля. На основе базового класса создается производный класс, у которого появляется еще одно текстовое поле. Свойство переопределяется так, что значением возвращается текст, получающийся объединением (через пробел) значений текстовых полей объекта. При присваивании значения свойству присваиваемая текстовая строка разбивается на две, которые присваиваются полям объекта. Разделителем для разбивки строки на две подстроки является пробел (первый с начала строки). Если пробела нет, то первая подстрока совпадает с исходной строкой, а вторая подстрока пустая. Метод `ToString()` для производного класса нужно переопределить таким образом, чтобы он возвращал название класса и значения полей объекта.

10. Напишите программу, в которой есть базовый класс с защищенным целочисленным массивом, индексатором (с целочисленным индексом), позволяющим считывать и присваивать значения элементам массива, а также свойство, возвращающее результатом размер массива. На основе базового класса создается производный класс, у которого появляется защищенный символьный массив. Опишите в производном классе версию индексатора с символьным индексом, который возвращает значение элемента символьного массива и позволяет присвоить значение элементу символьного массива. Для свойства из базового класса необходимо выполнить замещение так, чтобы результатом возвращался целочисленный массив из двух элементов: первый элемент определяет размер целочисленного массива объекта, а второй элемент определяет размер символьного массива объекта.

Заключение
ЧТО БУДЕТ ДАЛЬШЕ

Прекратите панику, Фукс. И слушайте мепя внимательпо!

из м/ф «Приключения капитана Врунгеля»

На этом первая часть книги закончена. Мы познакомились с основными, наиболее важными темами. Это основа, ядро языка C#. На данном этапе читатель уже вполне подготовлен к написанию сложных программ. Тем не менее точку в изучении языка C# мы не ставим. Впереди еще много полезного и интересного. Планы у нас большие, и их воплощению посвящена вторая часть книги.

Во второй части обсуждаются несколько фундаментальных концепций, которые в полной мере раскрывают красоту и мощь языка C#. Там мы познакомимся с интерфейсами и абстрактными классами. Узнаем, что такое делегаты, ссылки на методы, лямбда-выражения и анонимные методы, познакомимся с событиями. Познакомимся с перечислениями и выясним, чем структуры отличаются от классов. Мы научимся с помощью указателей выполнять операции с памятью. Узнаем, как в языке C# реализуется перехват и обработка ошибок. Для нас откроются секреты создания многопоточных приложений. Мы научимся использовать обобщенные типы и сможем создавать приложения с графическим интерфейсом. Еще будут операции с файлами, знакомство с коллекциями и многое другое. Так что работа предстоит большая и интересная.

Предметный указатель

Научно-популярное издание

РОССИЙСКИЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Алексей Васильев

ПРОГРАММИРОВАНИЕ НА C# ДЛЯ НАЧИНАЮЩИХ
Основные сведения

Директор редакции *Е. Капьёв*
Ответственный редактор *Е. Истомина*
Младший редактор *Е. Минина*
Художественный редактор *В. Брагина*

В оформлении обложки использована иллюстрация:
pavel7tymoshenko / Shutterstock.com
Используется по лицензии от Shutterstock.com

ООО «Издательство «Эксмо»
123308, Москва, ул. Зорге, д. 1. Тел.: 8 (495) 411-68-86.
Home page: **www.eksmo.ru** E-mail: **info@eksmo.ru**

Өндіруші: «ЭКСМО» АҚБ Баспасы, 123308, Мәскеу, Ресей, Зорге көшесі, 1 үй.
Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru.
Тауар белгісі: «Эксмо»
Қазақстан Республикасында дистрибьютор және өнім бойынша
арыз-талаптарды қабылдаушының
өкілі «РДЦ-Алматы» ЖШС, Алматы қ., Домбровский көш., 3«а», литер Б, офис 1.
Тел.: 8 (727) 251-59-90/91/92; E-mail: RDC-Almaty@eksmo.kz
Интернет-магазин: www.book24.kz
Өнімнің жарамдылық мерзімі шектелмеген.
Сертификация туралы ақпарат сайтта: www.eksmo.ru/certification

Сведения о подтверждении соответствия издания согласно законодательству РФ о техническом регулировании можно получить по адресу: http://eksmo.ru/certification/

Өндірген мемлекет: Ресей
Сертификация қарастырылмаған

Подписано в печать 26.03.2018. Формат 70x100$^{1}/_{16}$.
Печать офсетная. Усл. печ. л. 47,96.
Тираж экз. Заказ

ISBN 978-5-04-092519-3

Новая книга Алексея Васильева, ведущего российского автора самоучителей по программированию, посвящена языку C#. Один из самых популярных языков семейства C, широко используемый во всем мире, откроется вам во всем многообразии в этой первой книге двухтомника «Программирование на C# для начинающих». В ней доступным для новичков языком автор рассказывает читателям об основах этого языка: структура, операторы, циклы, массивы и многие другие сведения, нужные вам для того, чтобы начать программировать на C#.

Книга содержит множество примеров и подробный разбор каждого из них, а также задания для самостоятельной работы.

Прочитав этот самоучитель, вы будете обладать полным спектром сведений по языку C#, необходимых для того, чтобы приступить к самостоятельному программированию на нем.

Дополнительные материалы можно скачать по адресу: https://eksmo.ru/files/vasilyev_csharp.zip

Самое главное:

- Основные сведения о языке C# — от истории до создания небольших программ
- Подробный разбор каждой главы с примерами и выводами
- Все примеры актуальные и могут применяться в работе
- Доступный язык изложения, понятный новичкам
- Использована методика обучения, многократно проверенная на практике

Об авторе

Алексей Николаевич Васильев — доктор физико-математических наук, профессор кафедры теоретической физики физического факультета Киевского национального университета имени Тараса Шевченко. Автор более 15 книг по программированию на языках C++, Java, JavaScript, Python и математическому моделированию.

Большинство книг по программированию — скучное изложение спецификаций и примеров. С этой книгой начинающий программист может сразу приступить к написанию кода на языке C#, поскольку примеры в ней хорошо описаны и проиллюстрированы. Вы освоите минимальный набор инструментов, практику их применения, а задания для самостоятельной работы в конце тематических глав помогут проверить полученные знания.

Алексей Флоринский,
заместитель директора SimbirSoft

ISBN 978-5-04-092519-3