Г. Шилдт

C#

учебный курс

▸▸ ПРОГРАММИРОВАНИЕ

прекрасное пособие
по новому языку
программирования

bhv

OSBORNE

ПИТЕР

Г. Шилдт

C#

bhv® ПИТЕР®

Москва • Санкт-Петербург • Нижний Новгород • Воронеж
Ростов-на-Дону • Екатеринбург • Самара
Киев • Харьков • Минск
2003

Содержание

Предисловие ... 12
Структура книги ... 12
Опыт программирования не обязателен ... 13
Необходимое программное обеспечение ... 13
Не забудьте: код доступен через Internet ... 13
Для дальнейшего изучения ... 13
Об авторе ... 14

Глава 1. Основы языка C# ... 15
Генеалогическое древо языка C# ... 16
Создание языка C — начало современной эры программирования ... 16
Появление ООП и создание языка C++ ... 17
Internet и появление языка Java ... 18
История создания языка *C#* ... 19
Связь C# с .NET Framework ... 20
Что такое .NET Framework ... 20
Как работает не зависящая от языка среда исполнения ... 21
Управляемый и не управляемый коды ... 21
Общая языковая спецификация ... 22
Объектно-ориентированное программирование ... 22
Инкапсуляция ... 23
Полиморфизм ... 24
Наследование ... 25
Первая простая программа ... 25
Компилирование из командной строки ... 26
Использование Visual C++ IDE ... 27
Анализ первой программы ... 30
Обработка синтаксических ошибок ... 33
Небольшое изменение программы ... 33
Вторая простая программа ... 34
Другие типы данных ... 36
Проект 1-1. Преобразование значений температуры ... 38
Два управляющих оператора ... 39
Оператор if ... 39
Цикл for ... 41

Использование блоков кода....42
Символ точки с запятой и позиционирование.... 44
Использование отступов....44
Проект 1-2. Усовершенствование программы по преобразованию значения температуры 45
Ключевые слова в языке C#....46
Идентификаторы....46
Библиотека классов C#.... 47
Контрольные вопросы....48

Глава 2. Типы данных и операторы....49

Строгий контроль типов данных в C# 50
Обычные типы данных.... 50
Целочисленные типы....51
Типы данных с плавающей точкой....53
Тип decimal....54
Символы....55
Булев тип данных....56
Форматирование вывода.... 57
Проект 2-1. Разговор с Марсом 59
Литералы.... 61
Шестнадцатеричные литералы....62
Символьные escape-последовательности....62
Строковые литералы....63
Переменные и их инициализация....65
Инициализация переменной....65
Динамическая инициализация....66
Область видимости и время жизни переменных....66
Операторы....70
Арифметические операторы....70
Операторы сравнения и логические операторы....72
Проект 2-2. Вывод таблицы истинности логических операторов 75
Оператор присваивания....77
Составные операторы присваивания....78
Преобразование типа в операциях присваивания....79
Выполнение операции приведения типа между несовместимыми типами данных....80
Приоритетность операторов....82
Выражения....82
Преобразование типов в выражениях....82

Использование пробелов и круглых скобок....85
Проект 2-3. Вычисление суммы регулярных выплат по кредиту 86
Контрольные вопросы....89

Глава 3. Управляющие операторы....90

Ввод символов с клавиатуры....91
Оператор if....92
Вложенные операторы if....93
Цепочка операторов if-else-if....94
Оператор switch....96
Вложенный оператор switch....100
Проект 3-1. Построение простой справочной системы C#....101
Цикл for....103
Некоторые варианты цикла for....104
Недостающие части цикла for....105
Циклы, не имеющие тела....107
Объявление управляющих переменных цикла внутри цикла for....107
Цикл while....109
Цикл do-while....110
Проект 3-2. Совершенствование справочной системы C#....112
Использование оператора break для выхода из цикла....114
Использование оператора continue....117
Оператор goto....117
Проект 3-3. Завершение создания справочной системы C# 119
Вложенные циклы....122
Контрольные вопросы....124

Глава 4. Классы, объекты и методы....126

Основные понятия класса....127
Общий синтаксис класса....127
Определение класса....128
Как создаются объекты....132
Переменные ссылочного типа и оператор присваивания....133
Методы....134
Добавление метода к классу Vehicle....135
Возврат управления из метода 137
Возвращение методом значений....138
Использование параметров....141
Дальнейшее усовершенствование класса Vehicle....142

Проект 4-1. Создание справочного класса....144
Конструкторы....149
Конструкторы с параметрами....151
Добавление конструктора к классу Vehicle....151
Оператор new....153
Деструкторы и «сборка мусора»....153
Деструкторы....154
Проект 4-2. Демонстрация работы деструкторов....155
Ключевое слово this....157
Контрольные вопросы....159

Глава 5. Подробнее о типах данных и операторах....160
Массивы....161
Одномерные массивы....161
Проект 5-1. Сортировка массива....165
Многомерные массивы....167
Двухмерные массивы....167
Массивы, имеющие более двух размерностей....169
Инициализация многомерных массивов....169
Невыровненные массивы....170
Присваивание ссылок на массив....172
Использование свойства Length....173
Проект 5-2. Класс Queue....176
Цикл foreach....180
Строки....183
Создание объектов класса String....183
Операции со строками....184
Массивы строк....186
Неизменность строк....186
Побитовые операторы....188
Побитовые операторы AND, OR, XOR и NOT....188
Операторы сдвига....193
Составные побитовые операторы....194
Проект 5-3. Класс ShowBits....195
Оператор ?....197
Контрольные вопросы....200

Глава 6. Детальное рассмотрение методов и классов....201
Управление доступом к членам класса....202

Модификаторы в C#....202
Проект 6-1. Усовершенствованный класс Queue....208
Передача объектов методу....209
Как передаются аргументы....210
Использование параметров с модификаторами ref и out....212
Использование модификатора ref....213
Использование модификатора out....215
Использование переменного количества аргументов....217
Возвращение объектов....220
Перегрузка метода....222
Перегрузка конструкторов....228
Вызов перегруженного конструктора с использованием ключевого слова this....230
Проект 6-2. Перегрузка конструктора Queue....231
Метод Main()....234
Возвращение значений методом Main()....234
Передача аргументов методу Main()....235
Рекурсия....237
Ключевое слово static....239
Проект 6-3. Алгоритм Quicksort....242
Контрольные вопросы....245

Глава 7. Перегрузка оператора, индексаторы и свойства....246

Перегрузка операторов....247
Синтаксис метода оператора....247
Перегрузка бинарных операторов....248
Перегрузка унарных операторов....250
Дополнительные возможности класса ThreeD....254
Перегрузка операторов сравнения....258
Основные положения и ограничения при перегрузке операторов....260
Индексаторы....261
Многомерные индексаторы....266
Свойства....269
Ограничения в использовании свойств....272
Проект 7-1. Создание класса Set....273
Контрольные вопросы....282

Глава 8. Наследование....283

Основы наследования....284
Доступ к членам класса при использовании наследования....287

Использование модификатора protected..... 290
Конструкторы и наследование.....291
Вызов конструкторов наследуемого класса.....293
Скрытие переменных и наследование 297
Использование ключевого слова base для доступа к скрытой переменной.....298
Проект 8.1. Расширение возможностей класса Vehicle..... 300
Создание многоуровневой иерархии классов.....304
Когда вызываются конструкторы..... 307
Ссылки на объекты наследуемого и наследующего классов.....308
Виртуальные методы и переопределение.....313
Для чего нужны переопределенные методы.....315
Применение виртуальных методов.....316
Использование абстрактных классов.....320
Использование ключевого слова sealed с целью предотвращения наследования.....324
Класс object 325
Упаковка и распаковка.....326
Контрольные вопросы.....329

Глава 9. Интерфейсы, структуры и перечисления 330

Интерфейсы.....331
Реализация интерфейсов..... 332
Использование интерфейсных ссылок.....336
Проект 9.1. Создание интерфейса Queue.....338
Свойства интерфейса..... 343
Интерфейсные индексаторы.....344
Наследование интерфейсов.....346
Явная реализация.....348
Структуры.....350
Перечисления.....352
Инициализация перечисления.....354
Указание базового типа перечисления..... 354
Контрольные вопросы.....355

Глава 10. Обработка исключений.....356

Класс System. Exception.....357
Основы обработки исключений.....358
Использование блоков try и catch.....358
Простой пример исключения.....359
Второй пример исключения.....360

Если исключение не перехвачено....361
Изящная обработка ошибок с помощью исключений....363
Использование нескольких операторов catch.... 364
Перехват всех исключений....365
Вложенные блоки try.... 366
Генерирование исключений.... 367
Повторное генерирование исключения.... 368
Использование блока finally....369
Более близкое знакомство с исключениями....371
Часто используемые исключения....372
Наследование классов исключений....373
Перехват исключений производного класса.... 375
Проект 10-1. Добавление исключений в класс Queue....377
Использование ключевых слов checked и unchecked....379
Контрольные вопросы.... 383

Глава 11. Ввод/вывод....384
Потоки ввода/вывода C#....385
Байтовые потоки и потоки символов.... 385
Предопределенные потоки....385
Классы потоков.... 386
Класс Stream.... 386
Классы байтовых потоков....387
Классы потоков символов.... 388
Двоичные потоки....389
Консольный ввод/вывод.... 389
Чтение данных с консоли.... 390
Вывод данных на консоль....391
Класс FileStream и байт-ориентированный ввод/вывод в файлы....392
Открытие и закрытие файла....392
Чтение байтов с помощью класса FileStream....394
Запись в файл....395
Ввод/вывод в символьные файлы.... 397
Использование класса StreamWriter....398
Использование класса StreamReader.... 400
Перенаправление стандартных потоков.... 401
Проект 11-1. Утилита сравнения файлов....403
Чтение и запись двоичных данных....405
Поток BinaryWriter....405

Поток BinaryReader....406
Демонстрация двоичного ввода/вывода....407
Произвольный доступ к содержимому файла....409
Преобразование числовых строк в их внутренние представления....411
Проект 11.2. Создание справочной системы....415
Контрольные вопросы....421

Глава 12. Делегаты, события, пространства имен и дополнительные элементы языка C#....422
Делегаты....423
Многоадресность делегатов....427
Преимущества использования делегатов....429
События....430
Широковещательное событие....432
Пространства имен....434
Объявление пространства имен....435
Директива using....437
Вторая форма синтаксиса директивы using....439
Аддитивность пространств имен....440
Вложенные пространства имен....441
Пространство имен, используемое по умолчанию....443
Проект 12-1. Помещение класса Set в пространство имен....443
Операторы преобразования....446
Препроцессор....451
Директива препроцессора #define....451
Директивы препроцессора #if и #endif....452
Директивы препроцессора #elsc и #elif....453
Директива препроцессора #undef....455
Директива препроцессора #crror....456
Директива препроцессора #warning....456
Директива препроцессора #line....456
Директивы препроцессора #region и #endregion....457
Атрибуты....457
Атрибут Conditional....457
Атрибут Obsolete....459
Небезопасный код....460
Краткая информация об указателях....461
Ключевое слово unsafe....462
Ключевое слово fixed....463

Идентификаций типа но время работы программы....464
Проверка типа с помощью ключевого слова is....465
Ключевое слово as....466
Ключевое слово typeof....466
Другие ключевые слова....467
Модификатор доступа internal....467
Ключевое слово sizcof....467
Ключевое слово lock....467
Поле readonly....467
Ключевое слово stackalloc....468
Оператор using....469
Модификаторы const и volatile....469
Что делать дальше....469
Контрольные вопросы....470

Приложение. Ответы на контрольные вопросы.... -
Глава 1: Основы языка C#....-
Глава 2: Типы данных и операторы....-
Глава 3: Управляющие операторы.... -
Глава 4: Классы, объекты и методы.... -
Глава 5: Типы данных и операторы....-
Глава 6: Детальное рассмотрение методов и классов....-
Глава 7: Перегрузка операторов, индексаторы и свойства....-
Глава 8: Наследование....-
Глава 9: Интерфейсы, структуры и перечисления.... -
Глава 10: Обработка исключений.... -
Глава 11: Ввод/вывод.... -
Глава 12: Делегаты, события, пространства имен и дополнительные элементы языка C#.... -

Алфавитный указатель.... -

Предисловие

За пару последних лет компьютерный мир несколько сошел с проторенного пути, перейдя от концепции программирования для отдельных систем к созданию интерактивного сетевого окружения. Многие годы между компьютерными языками, операционными системами и средствами разработки велась борьба за рынок сбыта, но для большинства из них «сетью по-прежнему оставался одни компьютер». Теперь же перед современным языком программирования стоят совершенно новые задачи, для решения которых и был разработан язык C#, являющийся следующей ступенью в эволюции языков программирования. От своих предшественников он взял все самое лучшее, объединил новейшие разработки в области проектирования языков программирования. В первую очередь C# вобрал в себя лучшие качества C++ и Java — двух наиболее широко используемых языков. Кроме того, в него включены такие инновационные элементы, как делегаты (delegates) и индексаторы (indexers). А поскольку C# использует средства системы .Net Framework, его код является в высшей степени переносимым и может быть использован при разработке программных комплексов в многоязыковой среде. Компоненты программного обеспечения, созданные с использованием C#, совместимы с кодом, который написан на других языках, если он также предназначен для .Net Framework.

Основная задача книги — научить читателя основам программирования на C#. Для этого применяется пошаговый подход к изучению материала с использованием большого числа примеров и вопросов для самоконтроля, с разработкой множества проектов. Читателю необязательно иметь опыт программирования. Книга начинается с описания таких элементарных операций, как компилирование и запуск Сопрограмм. Затем в ней обсуждаются ключевые слова, функции и конструкции, которые собственно и составляют язык C#. Ознакомившись с изложенным здесь материалом, вы получите полное представление об основных принципах программировании на C#.

Важно понимать, что настоящее издание является всего лишь отправной точкой. C#-программирование — это не только ключевые слова и синтаксис, определяющий язык. Здесь речь должна идти и об использовании библиотеки .Net Framework Class Library, которая настолько обширна, что ее описание требует отдельной книги. Хотя несколько определяемых в библиотеке классов и обсуждается в книге, но по причине ограниченного ее объема о большинстве элементов библиотеки здесь даже не упоминается. Чтобы стать первоклассным программистом на C#, библиотеку следует изучить в совершенстве. Освоив данную книгу, вы получите необходимые знания и для освоения других аспектов языка C#.

И последнее: C# — это новый язык. Подобно всем новым компьютерным языкам, он должен пройти период совершенствования и становления. Читателям, надо полагать, будет интересно следить за появлением и развитием его новых свойств и возможностей. Не удивляйтесь, если он претерпит некоторые изменения. История языка C# только начинается.

Структура книги

Книгу следует рассматривать как учебное пособие, где каждая последующая глава базируется на предыдущей. Она состоит из 12 глав, в каждой из которых обсуждается отдельный аспект языка C#. А несколько специальных приемов и элементов структуры,

призванных прочно закрепить изучаемый материал в памяти читателей, делают книгу просто уникальной.

- Каждая глава начинается с перечисления рассматриваемых в ней тем.
- Завершается каждая глава разделом «Контрольные вопросы», позволяющим читателю самостоятельно проконтролировать, насколько полно усвоен материал. Ответы на задаваемые вопросы приведены в приложении.
- В конце каждого основного раздела главы имеется «Минутный практикум», предназначенный для проверки того, как читатель понимает излагаемые положения. Ответы на задаваемые здесь вопросы приведены в нижней части страницы.
- Под рубрикой «Ответы профессионала» (оформлена в виде вопроса-ответа) содержится дополнительная информация, а также интересные комментарии к обсуждаемым темам.
- Каждая глава содержит один или более проектов, призванных продемонстрировать, как на практике применить полученные знания. В основном это примеры реального кода, которые можно использовать в качестве отправной точки при разработке собственных программ.

Опыт программирования не обязателен

Книга не требует какого-либо опыта программирования. Даже если вам никогда прежде не приходилось заниматься таковым, вы без труда освоите излагаемый здесь материал. Но преобладающая часть читателей наверняка имеет хотя бы небольшой опыт программирования, в первую очередь на языках C++ и Java. А C#, как вы сможете убедиться, довольно тесно связан с обоими этими языками. Таким образом, если вы уже знаете C++ или Java, то гораздо быстрее изучите C#.

Необходимое программное обеспечение

Для компилирования и запуска программ, приведенных в этой книге, вам понадобится Visual Studio.NET 7 (или более поздняя версия), а также инсталлированная система .NET Framework.

Не забудьте: код доступен через Internet

Помните, что исходный код всех примеров и проектов, описанных в данной книге, можно бесплатно загрузить с Web-узла **www.osborne.com.**

Для дальнейшего изучения

Данная книга, как уже было отмечено, входит в большую серию книг о программировании, автором которых является Герберт Шилдт. Ниже приведен перечень изданий, которые могут вас заинтересовать.

Желающим получить более полные сведения о языке C# рекомендуем приобрести книгу

C#: The Complete Reference

Для тех, кто хочет изучить C++, особенно полезными окажутся следующие издания:

C++: The Complete Reference

C++: A Beginner's guide

Teach Yourself C++

C++ from the Ground Up

STL Programming from the Ground Up

The C/C++ Programming Annotated Archives

А чтобы изучить Java-программирование, можно обратиться к изданиям:

Java 2: A Beginner's guide

Java 2: The Complete Reference

Java 2: The Programmer's Reference

Тем, кто желает научиться писать программы для Windows, мы рекомендуем следующие книги Герберта Шилдта:

Windows 98 Programming from the Ground Up

Windows 2000 Programming from the Ground Up

MFC Programming from the Ground Up

The Windows Programming Annotated Archives

Если же вы хотите изучить язык C, являющийся основой всех современных языков программирования, обратитесь к книгам:

C: The Complete Reference

Teach Yourself C

Каждый, кому нужны более фундаментальные знания и полные ответы, может обратиться к Герберту Шилдту, признанному специалисту в области программирования.

Об авторе

Один из самых популярных авторов книг по программированию, Герберт Шилдт, считается ведущим специалистом по языкам C, C++, Java и C#, а также экспертом в области создания программ для Windows. Написанные им книги по программированию были переведены на многие языки и изданы общим тиражом более 3 миллионов экземпляров. Герберт Шилдт является автором огромного числа изданий, ставших бестселлерами. Наиболее известные среди них — это *C++: The Complete Reference, Java 2: The Complete Reference*; *Java 2: A Beginner's Guide*, *Windows 2000 Programming from the Ground Up* и *C: The Complete Reference.* Шилдт имеет степень мастера компьютерных наук, присвоенную ему университетом штата Иллинойс. Связаться с Гербертом можно через его консультационный офис по телефону (217) 548-4683.

Основы языка C#

- История создания языка C#
- Использование среды NET Framework
- Три принципа объектно-ориентированного программирования
- Создание, компилирование и запуск C#-программ
- Использование переменных
- Работа с операторами if и for
- Использование блоков кода
- Ключевые слова C#

Языки программирования служат самым разнообразным целям — от решения сложных математических задам и проведения экономико-математических расчетов до создания музыкальной партитуры и машинной графики. Попытки создания совершенного языка программирования предпринимаются столько же лет, сколько существует само программирование. В результате этого поиска компанией Microsoft был разработан язык C#, соответствующий современным стандартам программирования и предназначенный для поддержки развития технологии .NET Framework. Этот язык предоставляет эффективный метод написания программ для современных компьютерных систем предприятий, которые используют операционную систему Windows и компоненты Internet. Цель нашей книги — помочь вам научиться программировать на языке C#.

В этой главе мы расскажем об истории создания языка C# и опишем наиболее важные его свойства. Безусловно, изучение этого языка представляет определенные трудности, поскольку все его элементы тесно взаимосвязаны и не могут рассматриваться изолированно друг от друга. Чтобы справиться с этой проблемой, в данной главе мы кратко рассмотрим общую структуру C#-программ, а также основные управляющие элементы и условные операторы языка, сфокусировав внимание на базисных концепциях программирования, общих для любой C#-программы.

Генеалогическое древо языка C#

Компьютерные языки взаимосвязаны, а не существуют сами по себе. Каждый новый язык в той или иной форме наследует свойства ранее созданных языков, то есть осуществляется принцип преемственности. В результате возможности одного языка используются другими (например, новые характеристики интегрируются в уже существующий контекст, а старые конструкции языка удаляются). Так происходит эволюция компьютерных языков и совершенствуется искусство программирования.

Язык C# не является исключением, он унаследовал много полезных возможностей от других языков программирования и напрямую связан с двумя наиболее широко применяемыми в мире компьютерными языками — C и C++, а также с языком Java. Для понимания C# следует разобраться в природе такой связи, поэтому сначала мы расскажем об истории развития этих трех языков.

Создание языка C — начало современной эры программирования

Язык C был разработан Дэннисом Ричи, системным программистом из компании Bell Laboratories, город Мюррей-хилл, штат Нью-Джерси, в 1972 году. Этот язык настолько хорошо зарекомендовал себя, что в конечном счете на нем было написано более 90 % кода ядра операционной системы Unix (которую поначалу писали на языке низкого уровня — ассемблере). К моменту появления C более ранние языки, самым известным из которых является Pascal, использовались достаточно успешно, но именно язык C определил начало современной эры программирования.

Революционные изменения в технологии программирования, приведшие к появлению *структурного программирования* 1960-х годов, обусловили базовые возможности для создания языка C. До появления структурного программирования трудно было писать большие программы, поскольку с увеличением количества строк код программы

превращался в запутанную массу переходов, вызовов и возвращаемых результатов, за которыми сложно было проследить ход самой программы. Структурные языки решили эту проблему, добавив в инструментарий программиста условные операторы, процедуры с локальными переменными и другие усовершенствования. Таким образом появилась возможность писать относительно большие программы.

Именно язык С стал первым структурным языком, в котором успешно сочетались мощность, элегантность и выразительность. Такие его свойства, как краткость и легкий в использовании синтаксис в сочетании с принципом, согласно которому ответственность за возможные ошибки возлагается на программиста, а не на язык, быстро нашли множество сторонников. Сегодня мы считаем эти качества само собой разумеющимися, но тогда в С впервые были воплощены великолепные новые возможности, так необходимые программистам. В итоге с 1980-х годов С является самым используемым языком структурного программирования.

Но по мере развития программирования появляется проблема обработки программ все большего размера. Как только код проекта достигает определенного объема (его числовое значение зависит от программы, программиста, используемых инструментов, но приблизительно речь идет о 5000 строк кода) возникают трудности в понимании и сопровождении С-программы.

Появление ООП и создание языка С++

В конце 1970-х годов настал момент, когда многие проекты достигли максимального размера, доступного для обработки с помощью языка структурного программирования С. Теперь требовались новые подходы, и для решения этой проблемы было создано *объектно-ориентированное программирование* (ООП), позволяющее программисту работать с программами большего объема. А поскольку С, являвшийся в то время самым популярным языком, не поддерживал ООП, возникла необходимость создания его объектно-ориентированной версии (названной позднее С++).

Эта версия была разработана в той же компании Bell Laboratories Бьярном Страуструпом в начале 1979 года. Первоначально новый язык получил название «С с классами», но в 1983 году был переименован в С++. Он полностью включает в себя язык С (то есть С служит фундаментом для С++) и содержит новые возможности, предназначенные для поддержки объектно-ориентированного программирования. Фактически С++ является объектно-ориентированной версией языка С, поэтому программисту, знающему С, при переходе к программированию на С++ надо изучить только новые концепции ООП, а не новый язык.

Долгое время язык С++ развивался экстенсивно и оставался в тени. В начале 1990-х годов он начинает применяться массово и приобретает огромную популярность, а к концу десятилетия становится наиболее широко используемым языком разработки программного обеспечения, лидирующим и сегодня.

Важно понимать, что разработка С++ не является попыткой создания нового языка программирования, а лишь совершенствует и дополняет уже достаточно успешный язык. Такой подход, ставший новым направлением развития компьютерных языков, используется и сейчас.

Internet и появление языка Java

Следующим большим достижением в развитии языков программирования стал язык Java. Работа над Java, который изначально назывался Oak, началась в 1991 году в компании Sun Microsystems. Основными разработчиками этого языка были Джеймс Гослинг, Патрик Нотон, Крис Ворт, Эд Франк и Майк Шеридан.

Java является структурным объектно-ориентированным языком с синтаксисом и стратегией, взятыми из языка C++. Инновации Java были обусловлены бурным развитием информационных технологий и стремительным увеличением количества пользователей Internet, а также совершенствованием технологии программирования. До широкого распространения Internet большинство написанных программ компилировались для конкретных процессоров и определенных операционных систем. И хотя программисты при написании удачной программы практически всегда задавались вопросом повторного использования кода, этот вопрос не был первоочередным. Однако с развитием Internet (то есть появлением возможности соединения через сеть компьютеров с различными процессорами и операционными системами) на первый план вышла именно проблема легкого переноса программ с одной платформы на другую. Для решения этой задачи необходим был новый язык, которым и стал Java.

Нужно отметить, что сначала Java отводилась более скромная роль, он создавался как не зависящий от платформы язык, который можно было бы применять при создании программного обеспечения для встроенных контроллеров. В 1993 году стало очевидным, что технологии, использовавшиеся для решения проблемы переносимости в малом масштабе (для встроенных контроллеров), можно использовать в большом масштабе (для Internet). Самая главная возможность Java — способность создания межплатформенного переносимого кода — и стала причиной быстрого распространения этого языка.

В Java переносимость достигается посредством транслирования исходного кода программы в промежуточный язык, называемый *байт-кодом,* который затем выполняется виртуальной машиной Java (Java Virtual Machine, JVM). Следовательно, Java-программа может быть запущена на любой платформе, имеющей JVM. А поскольку JVM относительно легко реализуется, она доступна для большого числа платформ.

Использование байт-кода в Java радикально отличается от применения кодов в языках C и C++, практически всегда компилируемых в исполняемый машинный код. Машинный код связан с определенным процессором и операционной системой, следовательно, для запуска C/C++-программы на других платформах необходимо перекомпилировать исходный код программы в машинный код каждой их этих платформ (то есть нужно иметь несколько различных исполняемых версий программы). Понятно, что это трудоемкий и дорогой процесс. А в Java было предложено элегантное и эффективное решение — использование промежуточного языка. И это же решение в дальнейшем было применено в C#.

Уже говорилось, что в соответствии с новым подходом авторы Java создали его на основе C и C++ (синтаксис Java базируется на C, а объектная модель развилась из C++). Хотя Java-код не совместим с C или C++, его синтаксис сходен с синтаксисом этих языков, поэтому большая часть программистов, использовавших C и C++, смогла перейти на Java без особых усилий. Как Страуструпу при разработке C++, так и авторам Java не понадобилось создавать совершенно новый язык, они использовали

в качестве базовых уже известные языки и смогли сосредоточить внимание на инновационных элементах. Стоит отмстить, что после появления Java языки C и C++ стали общепринятым фундаментом для создания новых компьютерных языков.

История создания языка C#

Хотя язык Java решил многие проблемы переносимости программ с одной платформы на другую, все же для успешной работы в современном Internet-окружении ему недостает некоторых свойств, одним из которых является поддержка возможности взаимодействия нескольких компьютерных языков (многоязыкового программирования). Под *многоязыковым программированием* понимается способность кода, написанного на разных языках, работать совместно. Эта возможность очень важна при создании больших программ, а также желательна при программировании отдельных компонентов, которые можно было бы использовать во многих компьютерных языках и в различной операционной среде.

Серьезным недостатком Java является отсутствие прямой поддержки платформы Windows, являющейся сегодня наиболее широко используемой операционной системой в мире. (Хотя Java-программы могут выполняться в среде Windows при наличии инсталлированной JVM.)

Чтобы решить эти проблемы, компания Microsoft в конце 1990-х годов разработала язык C# (главный архитектор языка Андерс Хейльсберг), являющийся составной частью общей стратегии .NET этой компании. Альфа-версия языка была выпущена в середине 2000 года.

Язык C# напрямую связан с широко применяемыми и наиболее популярными во всем мире языками программирования C, C++ и Java. Сегодня практически все профессиональные программисты знают эти языки, поэтому переход к базирующемуся на них C# происходит без особых трудностей. Хейльсберг, так же как авторы языков C++ и Java, не «изобретал колесо», а пошел по проторенному пути — используя в качестве фундамента ранее созданные языки, сосредоточился на улучшениях и инновациях.

Генеалогическое древо C# показано на рис. 1.1. Язык C# строится на объектной модели, которая была определена в C++, а синтаксис, многие ключевые слова и операторы он унаследовал от языка C. Если вы знаете эти языки программирования, то у вас не возникнет проблем при изучении C#.

Связь между C# и Java более сложная. Оба языка разработаны для создания переносимого кода, базируются на C и C++, используют их синтаксис и объектную модель. Однако между этими языками нет прямой связи, они больше похожи на двоюродных братьев, имеющих общих предков, но отличающихся многими признакам. Если вы умеете работать на Java, это облегчит вам освоение C#, и наоборот, при изучении Java вам пригодится знание многих концепций C#.

Язык C# содержит многие инновационные свойства, которые будут рассматриваться в этой книге. Сразу заметим, что некоторые из его наиболее важных нововведений относятся к встроенной поддержке компонентов программного обеспечения. То есть фактически C# создан как компонентно-ориентированный язык, включающий, например, элементы (такие как свойства, методы и события), непосредственно поддерживающие составные части компонентов программного обеспечения. Но

пожалуй, наиболее важная новая характеристика C# — способность работать в многоязыковом окружении.

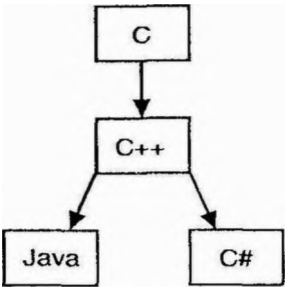


Рис. 1.1. Генеалогическое древо C#

Связь C# с .NET Framework

Хотя C# как язык программирования может изучаться изолированно, лучше рассматривать его во взаимосвязи с .NET Framework — средой, в которой он работает. Потому что, во-первых, C# изначально разрабатывался компанией Microsoft для создания кода .NET Framework, во-вторых, .NET Framework определяет библиотеки, используемые языком C#. Поскольку они так тесно связаны, важно понимать общую концепцию .NET Framework и ее значимость для C#.

Что такое .NET Framework

Отвечая на вопрос, вынесенный и заголовок, видимо, можно сказать, что .NET Framework определяет среду, которая поддерживает развитие и выполнение платформонезависимых приложений. Она допускает совместную работу в приложении различных языков программирования, а также обеспечивает переносимость программ и общую модель программирования для Windows. Важно отмстить, что .NET Framework не ограничена платформой Windows и написанные для этой платформы программы могут быть в будущем перенесены на другие платформы.

Язык C# использует две важные составляющие .NET Framework. Первая — это *не зависящая от языка среда исполнения* (Common Language Runtime, CLR), система, управляющая исполнением вашей программы, и являющаяся частью технологии .NET Framework, которая позволяет программам быть переносимыми, поддерживает программирование с использованием нескольких языков и обеспечивает безопасность передачи данных.

Вторая составляющая — *библиотека классов* .NET, которая дает программе доступ к среде исполнения, например, используется для ввода/вывода данных. Если вы начинающий программист, то вам может быть не знаком термин *масс.* Подробно мы расскажем о классах немного позже, а сейчас просто скажем, что классом является объектно-ориентированная конструкция, которая помогает организовать программу. Пока ваша программа ограничена характеристиками, определенными библиотекой классов .NET, она может быть запущена везде, где поддерживается среда исполнения .NET.

Как работает не зависящая от языка среда исполнения

Не зависящая от языка среда исполнения (CLR) управляет выполнением кода .NET. Расскажем, как она работает. При компилировании C#-программы мы получаем не исполняемый код, а файл, содержащий специальный тип псевдокода, называемый промежуточным языком Microsoft (Microsoft Intermediate Language, MSIL). Язык MSIL определяет набор переносимых инструкций, не зависимых от конкретного процессора, то сеть, по существу, MSIL определяет переносимый язык ассемблера. Обратим ваше внимание на то, что концептуально промежуточный язык Microsoft похож на байт-код Java, но между ними имеются различия.

Система CLR транслирует промежуточный код в исполняемый во время запуска программы. Любая программа, компилированная в MSIL, может быть запущена в любой операционной системе, для которой реализована среда CLR. Это часть механизма, с помощью которого в .NET Framework достигается переносимость программ.

Промежуточный язык Microsoft превращается в исполняемый код при использовании JIT-компилятора (англ. just in time — в нужный момент). Процесс работает следующим образом: при выполнении .NET-программы система CLR активизирует JIT-компилятор, который затем превращает MSIL во внутренний код данного процессора, причем коды частей программы преобразуются по мере того, как в них возникает потребность. Следовательно, ваша Сопрограмма фактически исполняется как внутренний код, хотя изначально она компилировалась в MSIL. Это означает, что время запуска нашей программы практически такое же, как если бы она была сразу скомпилирована во внутренний код, но при этом у вас появляется преимущество MSIL — переносимость программы.

Кроме того, при компилировании C#-программы в дополнение к MSIL вы получаете еще один компонент — *метаданные*, которые описывают данные, используемые вашей программой, и позволяют вашему коду взаимодействовать с другим кодом. Метаданные содержатся в том же файле, что и MSIL.

В принципе этих знаний о среде CLR, MSIL и метаданных достаточно для реализации основной части задач программирования на языке C#, поскольку этот язык самостоятельно организует работу надлежащим образом.

Управляемый и не управляемый коды

В целом, когда вы пишите C#-программу, то создаете так называемый *управляемый код*, который выполняется под контролем не зависящей от языка среды исполнения (CLR). Поскольку программа запускается под контролем CLR, управляемый код должен соответствовать определенным требованиям, при выполнении которых он получает множество преимуществ, включая современное управление памятью, способность совмещать языки, высокий уровень безопасности передачи данных, поддержку контроля версии и понятный способ взаимодействия компонентов программного обеспечения. Требования, которым должен соответствовать управляемый код, следующие: компилятор должен создать MSIL-файл, предназначенный для CLR, а также использовать библиотеки .NET Framework (и то, и другое делает C#).

Альтернативой управляемому коду является *не управляемый код,* который не выполняется CLR. До появления .NET Framework все Windows-программы использовали не управляемый код, сейчас же оба вышеназванных кода могут работать вместе, и язык C#, генерируя управляемый код, способен взаимодействовать с созданными ранее программами.

Общая языковая спецификация

Все преимущества управляемого кода обеспечиваются CLR. Если же ваш код будет использоваться программами, написанными на других языках, для максимальной совместимости необходимо придерживаться общей языковой спецификации (Common Language Specification, CLS). Эта спецификация описывает набор характеристик, являющихся общими для различных языков. Соответствие кода обшей языковой спецификации особенно важно при создании компонентов программного обеспечения, которые будут использоваться другими языками. Если и будущем нам придется создавать коммерческий код, то вы должны будете придерживаться спецификации CLS, поэтому мы привели информацию о ней в данном разделе, хотя это и не имеет непосредственного отношения к нашей книге.

Ответы профессионала

Вопрос. Зачем для решения вопросов переносимости, безопасности и программирования с использованием нескольких языков нужно было создавать новый компьютерный язык C#? Нельзя ли было адаптировать язык C++ для поддержки .NET Framework?

Ответ. Да, язык C++ можно адаптировать так, чтобы он создавал .NET-совместимый код, запускаемый под управлением среды CLR. Первоначально компания Microsoft так и поступила, добавив к C++ так называемые *управляемые расширения,* которые сделали возможным перенос существующего кода в .NET Framework. Но новая разработка .NET позволяет намного легче выполнить это в C#. Помните, что язык C# специально разрабатывался для платформы .NET и предназначен для поддержки развития этой технологии.

Минутный практикум

1. С какими языками язык C# имеет непосредственные связи?
2. Что такое не зависящая от языка среда исполнения (CLR)?
3. Что такое JIT-компилятор?

Объектно-ориентированное программирование

Язык C# базируется на принципах объектно-ориентированного программирования (ООП), и все C#-программы являются в какой-то степени объектно-ориентированными. Поэтому для написания даже самой простой C#-программы очень важно знать принципы ООП.

ООП является мощной технологией выполнения задач, с которыми приходится сталкиваться программистам. Со времен изобретения компьютера методы программирования значительно изменились. На протяжении развития компьютерной науки

1. Язык *C#* является потомком C и C++, а также имеет родство с Java.
2. Не зависящая от языка среда исполнения (CLR) управляет выполнением .NET-программ.
3. JIT-компилятор преобразует MSIL-код во внутренний код данного процессора, причем коды частей программы преобразуются по мере того, как в них возникает потребность.

специалистам в основном приходилось решать вопросы, связанные с увеличением сложности программ. Например, на первых компьютерах программирование осуществлялось посредством изменения двоичных машинных инструкций, при этом использовались элементы управления передней панели компьютера. Такой метод хорошо работал, пока объем программы ограничивался несколькими сотнями инструкций. Когда же объем используемого кода вырос, был изобретен язык ассемблера, и программист, используя символическое представление машинных инструкций, уже смог работать с более сложными и громоздкими программами. Затем, приспосабливаясь к росту объема и сложности программ, программисты разработали высокоуровневые языки, такие как FORTRAN и COBOL. Когда же эти ранние языки достигли предела своих возможностей, было изобретено структурное программирование.

Заметьте, что на каждом этапе развития программирования создавались технологии и инструменты, позволяющие программисту решать все более сложные задачи. С каждым шагом на этом пути новые технологии вбирали в себя все самое лучшее от предыдущих. Настал момент, когда многие проекты приблизились к той границе, где структурное программирование уже не могло соответствовать предъявляемым требованиям, и возникла необходимость в принципиально новой прогрессивной технологии, которой и стало объектно-ориентированное программирование.

ООП вобрало в себя лучшие идеи структурного программирования и объединило их с несколькими новыми концепциями, в результате чего появился новый способ организации программ. Не вдаваясь в подробности, можно сказать, что программа составляется одним из двух способов: вокруг своего кода или вокруг своих данных. При использовании технологии структурного программирования программы обычно организуются вокруг кода. Такой метод можно рассматривать как «код, воздействующий на данные».

В ООП программы работают иным способом. Они организованы вокруг данных, и их ключевой принцип можно сформулировать как «контролируемый данными доступ к коду». В объектно-ориентированном языке вы определяете данные, а также процедуры, которые могут воздействовать на эти данные (то есть именно тип данных задаст виды операций, которые могут быть применены к этим данным).

Для поддержки принципов объектно-ориентированного программирования у всех языков ООП, включая C#, есть три общие черты — инкапсуляция, полиморфизм и наследование. Давайте рассмотрим их в отдельности.

Инкапсуляция

Инкапсуляция — это механизм программирования, связывающий воедино код и данные, которыми он манипулирует, а также ограждающий их от внешнего доступа и неправильного применения. В объектно-ориентированном языке код и все необходимые данные могут связываться таким способом, что создается автономная структура — объект. Другими словами, объектом является структура, поддерживающая инкапсуляцию.

В пределах объекта код, данные или и код, и данные могут быть либо закрытыми для Других объектов (`private`), либо открытыми (`public`).

Закрытые код и данные известны и доступны из другой части только этого же объекта (то есть к закрытому коду и данным не может быть осуществлен доступ из той части программы, которая существует за пределами данного объекта). Когда же код и

данные открыты (объявлены в рамках какого-либо объекта как `public`), другие части программы тоже имеют возможность с ними работать. Обычно части объекта,
объявленные как `public`, используются для обеспечения контролируемого интерфейса с закрытыми элементами объекта.

В C# основной структурой, использующей инкапсуляцию, является *класс*, специфицирующий как данные, так и код, который будет работать с этими данными. Язык C# применяет спецификацию классов для конструирования *объектов,* являющихся экземплярами классов. Следовательно, класс фактически является набором инструк-ций, в которых указано, как должен быть создан объект.

Код и составляющие класс данные называются *членами* класса, а данные, определенные классом, называются *переменными экземпляра.* Фрагменты кода, которые рабо-тают с этими данными, называются *методами члена класса* или просто *методами.* В C# термин *метод* принят для подпрограммы, которую в C/C[J]-+ программисты называют *функцией* (поскольку C# является прямым наследником C++, то иногда используется и термин *функция).*

Полиморфизм

Полиморфизм (от греческого «множество форм») является свойством, которое по-зволяет нескольким объектам, имеющим некоторые общие характеристики, получать доступ к одному интерфейсу и реализовывать упомянутые в нем методы. В качестве простого примера рассмотрим руль автомобиля. Руль (интерфейс) остается одинако-вым независимо от того, какой тип рулевого механизма фактически используется. То есть принцип работы руля всегда остается неизменным (например, поворот рулевого колеса влево всегда приводит к повороту автомобиля влево), хотя на автомобиле может быть установлено ручное управление или рулевое управление с усилителем. Преимуще-ство унифицированного интерфейса состоит в том, что если вы один раз научились водить автомобиль, то сможете управлять любым типом автомобиля.

Тот же принцип может быть применен в программировании. Например, рассмотрим стек — список типа LIFO (last in, first out). Вы можете работать с программой, которой требуется три стека различных типов (один для целочисленных значений, один для значений с плавающей запятой и один для символов). В этом случае для реализации каждого стека используется один и тот же алгоритм, хотя хранящиеся данные различны. В не объектно-ориентированном языке вам понадобилось бы создавать три различных набора стековых процедур с собственным именем для каждого набора. А в C# благодаря полиморфизму можно создать общий набор стековых процедур, которые подойдут для работы со всеми тремя типами стеков. Таким образом, определив методы, использующие один тип стека, вы сможете применять их и для остальных типов.

Обобщенно концепцию полиморфизма можно выразить так: «один интерфейс — множество методов». Это означает, что для группы похожих процессов можно создать унифицированный интерфейс. Полиморфизм позволяет уменьшать сложность программ путем использования единого интерфейса для спецификации *общего класса действий.* Применительно к каждой ситуации компилятор сам выбирает *специфиче-ское действие* (то есть метод), избавляя вас от необходимости делать это вручную. Вы только должны помнить, какие методы упомянуты в данном интерфейсе, и реализо-вывать их.

Наследование

Наследование — это свойство, с помощью которого один объект может приобретать свойства другого. При этом поддерживается концепция иерархической классифика-ции, имеющей направление сверху вниз. Например, можно сказать, что класс *Вкусное_Красное_яблоко* является частью класса *яблоки,* который в свою очередь является частью класса *фрукты*, принадлежащего классу *продукты.* То есть класс *продукты* обладает определенными качествами (съедобные, питательные и так далее), которые также применимы к его подклассу *фрукты.* В дополнение к этим качествам класс *фрукты* имеет свои специфические характеристики (сочные, сладкие и так далее), отличающие его от других продуктов. Класс *яблоки* определяет качества, которые присущи яблокам (растут на дереве, не являются тропическими фруктами и так далее). Класс *Вкусное_Красное_яблоко* в свою очередь наследует все качества предшествующих классов и определяет только те качества, которые делают его уникальным.

Без использования наследования каждый объект должен явно определять все свои характеристики; используя наследование, объект должен определить только те качества, которые делают его уникальным в пределах своего класса. Он может наследовать общие атрибуты от своих родительских классов. Следовательно, именно механизм наследования дает возможность одному объекту быть специфическим экземпляром своего класса.

Минутный практикум

1. Назовите принципы ООП.
2. Что является базовым элементом инкапсуляции?

Ответы профессионала

Вопрос. Вы утверждаете, что ООП является эффективным способом управления большими программами, но создается впечатление, что он может излишне усложнить относительно небольшие программы. Поскольку вы говорите, что все C#-программы являются в какой-то степени объектно-ориентированными, не возникнут ли неудобства при написании небольших C#-программы?

Ответ. Нет, дальше вы увидите, что для небольших объемов кода использование ООП в C# практически не отражается на сложности программ. Хотя C# следует строгой объектной модели, вы имеете достаточную свободу в определении степени ее применения. Для небольших программ «объектная ориентированность» едва ощутима, по мере увеличения объема программы в нее можно интегрировать большее количество объектно-ориентированных свойств.

Первая простая программа

Перед началом углубленного изучения деталей языка скомпилируем и запустим короткую и простую программу, написанную на C#.

```
/*
   Это простая программа, написанная на C#.
```

1. Принципами ООП являются инкапсуляция, полиморфизм и наследование.
2. Базовым элементом инкапсуляции является класс.

```
   Файл, содержащий код программы, — Example.cs.
*/

using System;

class Example {

  // C#-программа начинается с вызова метода Main ().
  public static void Main () {
    Console.WriteLine ("Простая C#-программа.");
  }
}
```

Существует два способа компилирования, запуска и редактирования C#-программ. Во-первых, вы можете выполнить компиляцию из командной строки, запустив программу `csc.exe`. Во-вторых, можете использовать интегрированную среду разработки Visual C++ (Integrated Development Environment, IDE). Оба эти способа описаны ниже.

Компилирование из командной строки

Компилирование из командной строки представляет собой более легкий способ работы с большинством из приведенных в книге C#-программ, хотя для коммерческих проектов вы, вероятно, будете использовать интегрированную среду разработки Visual C++. Для компилирования и запуска C#-программ из командной строки вам необходимо произвести три действия.

1. Создать файл, содержащий код программы.
2. Компилировать программу.
3. Запустить программу.

Создание файла

Представленные в этой книге программы можно загрузить с Web-узла компании Osborne **www.osborne.com** или при желании ввести код программы вручную. При ручном вводе загрузите программу в файл, используя текстовый редактор (например Notepad). Текстовые ASCII-файлы должны быть созданы без элементов форматирования, поскольку такая информация может привести к ошибкам при компилировании. После ввода программы присвойте файлу имя `Example.cs`.

Компилирование программы

Для компилирования программы необходимо запустить компилятор C# `csc.exe`, указав имя исходного файла в командной строке:

```
C:\>csc Example.es
```

Компилятор `csc` создает файл `Example.exe`, содержащий MSIL-версию программы. Хотя MSIL не является исполняемым кодом, он содержится в исполняемом `(exe)` файле. При попытке запуска файла `Example.exe` не зависящая от языка среда исполнения (CLR) автоматически активизирует JIT-компилятор. Но если вы попытаетесь выполнить файл `Example.exe` (или любой другой файл, содержащий MSIL) на компьютере, где не инсталлирована .NET Framework, программа не будет работать, поскольку отсутствует CLR.

Примечание

Возможно, перед запуском компилятора `csc.exe` вам нужно будет запустить командный файл `vcvars32.bat`, который обычно находится в каталоге ***\Program Files\Microsoft Visual Studio.NET\Vc7\Bin***. В качестве альтернативы можно активизировать процесс компилирования из командной строки, выбрав элемент **Visual Studio.NET Command Prompt** из списка инструментов, приведенных в подменю **Microsoft Visual Studio.NET 7.0,** находящемся в меню **Пуск | Программы** на панели задач.

Запуск программы

Для запуска программы введите ее имя в командной строке, как показано ниже:

```
C:\>Example
```

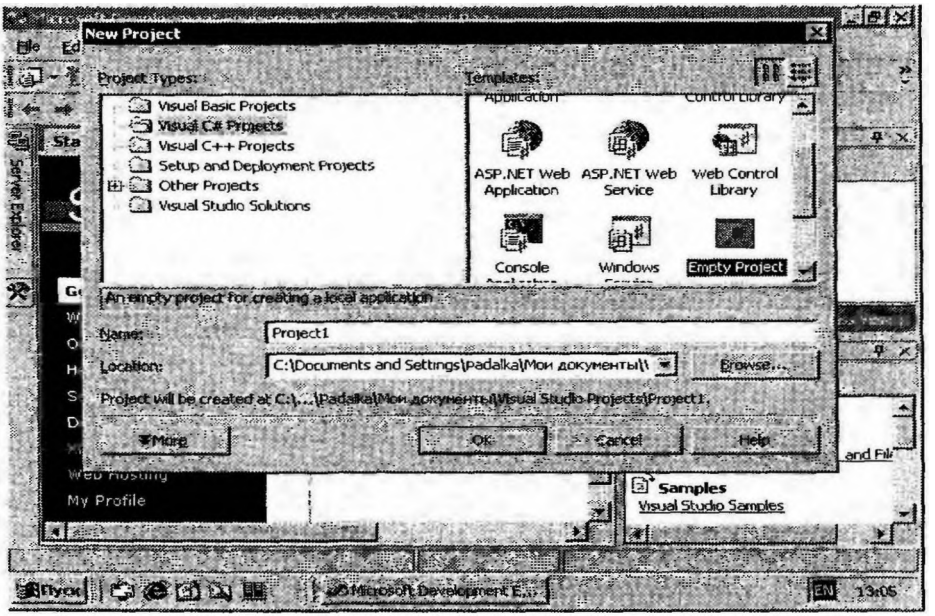
Когда программа будет запущена, на экране появится следующая строка:

```
Простая C#-программа.
```

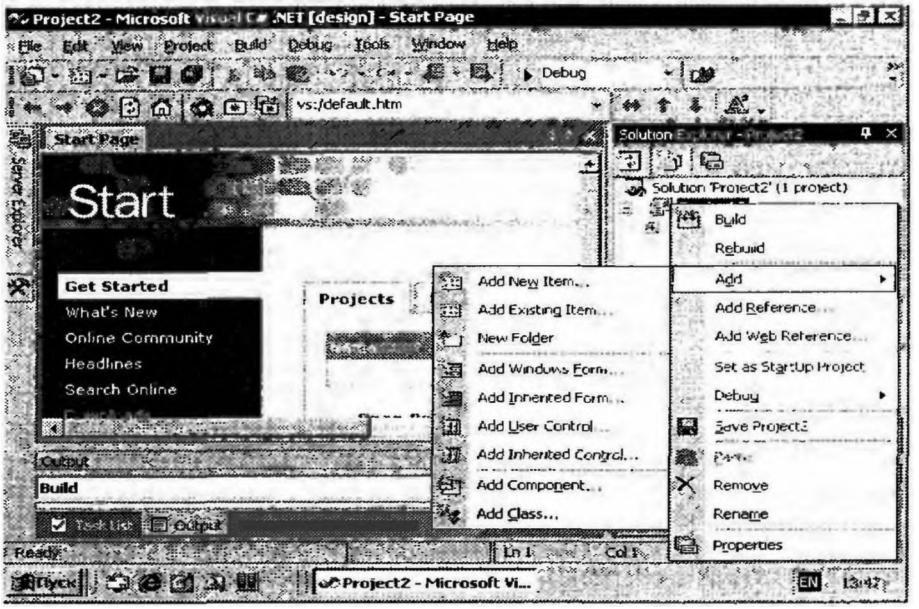
Использование Visual C++ IDE

Начиная с версии Visual Studio 7, Visual C++ IDE может компилировать C#-программы. Для редактирования, компилирования и запуска C#-программ с использованием Visual C++ IDE вам необходимо произвести действия, перечисленные ниже. (Если на вашем компьютере инсталлирована другая версия Visual C++, возможно, порядок действий будет другим.)

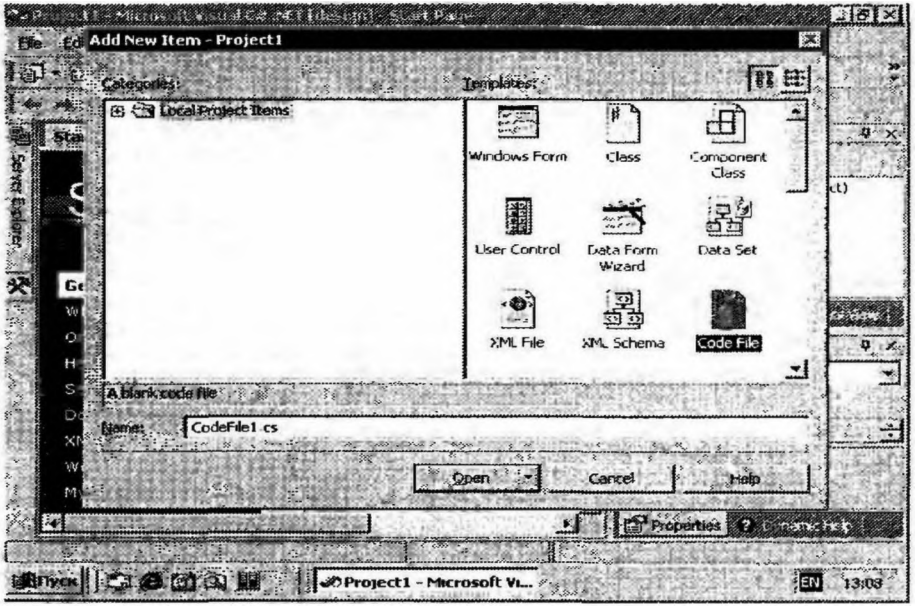
1. Создайте новый пустой C#-проект, выбрав пункты меню **File | New | Project.** Далее выберите папку **Visual C# Project,** а затем пиктограмму **Empty Project**, как показано на рисунке.



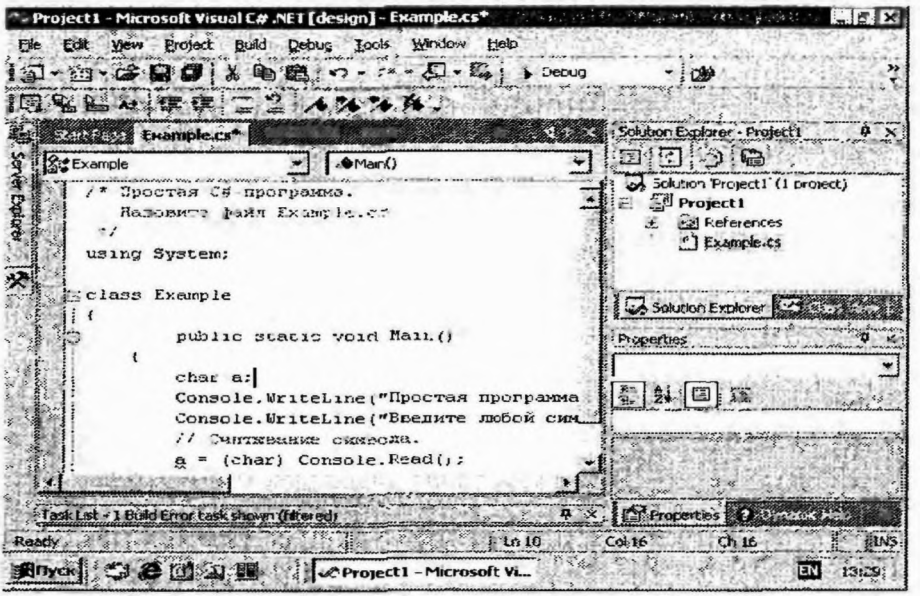
2. После создания проекта щелкните правой кнопкой мыши на имени проекта в окне **Solution**. В появившемся контекстном меню выберите пункт **Add**, затем пункт **Add New Item**. На экране вашего монитора появится следующая картинка.



3. Когда откроется диалоговое окно **Add New Item**, выберите пункт **Local Project Items**, затем пиктограмму **C# Code File**. Изображение на экране вашего монитора будет следующим.



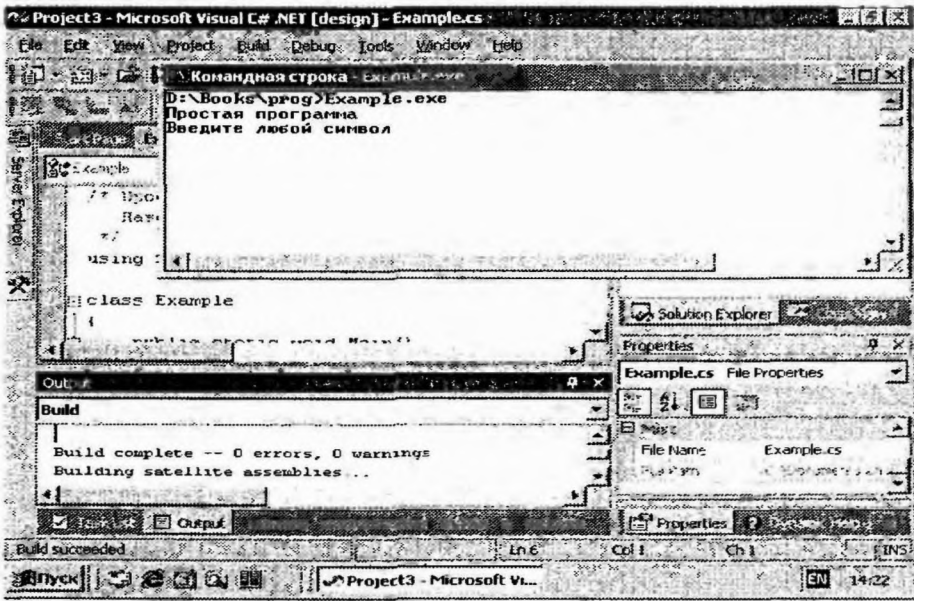
4. Загрузите программу и сохраните файл под именем `Example.cs`. (Помните, что программы, приведенные в этой книге, можно загрузить с Web-узла **`www.osborne.com`**.) На экране вашего монитора появится следующее изображение.



5. Скомпилируйте программу, выбрав пункт **Build** из меню **Build.**

6. Запустите программу, выбрав пункт **Start Without Debugging** из меню **Debug.**

Когда программа будет запущена, появится окно, показанное на рисунке.



Для компилирования и запуска программ, приводимых в этой книге, не обязательно каждый раз создавать новый проект, можно использовать тот же C#-проект. Для этого просто удалите текущий файл и добавьте новый, затем скомпилируйте его и запустите.

Как уже говорилось ранее, короткие программы, представленные в первой части книги, проще компилировать из командной строки.

Анализ первой программы

Хотя программа `Example.cs` достаточно короткая, она включает несколько ключевых характеристик, свойственных всем C#-программам. Давайте рассмотрим каждую часть программы, начиная с ее имени, более внимательно.

В отличие от некоторых языков программирования (например, Java), в которых очень важно правильно назвать файл, содержащий код программы, C# позволяет выбрать имя программы произвольно. Мы предложили назвать нашу программу `Example.cs`, но для C# было бы приемлемо и любое другое имя. Например, предыдущий файл программы можно назвать `Sample.es`, `Test.es` или даже `X.cs`.

По соглашению файлы C#-программ имеют расширение `.cs`. Но многие программисты называют файл по имени главного класса, определенного в рамках файла, вот почему мы выбрали для файла имя `Example.es`. Поскольку имя C#-программы выбирается произвольно, для большинства программ в этой книги мы не указали имена файлов, предоставив сделать это вам.

Программа начинается со следующих строк:

```
/*
   Это простая программа, написанная на C#.
   Файл, содержащий код программы, - Example.cs.
*/
```

Эти строки являются *комментарием*. C#, как и многие другие языки программирования, позволяет вводить комментарии в исходный код программы, при этом компилятор игнорирует содержимое комментария. В комментарии, адресованном тому, кто читает исходный код, описывается или объясняется операция, выполняемая в программе. В данном случае комментарий описывает программу и напоминает, что исходный файл должен быть назван `Example.cs`. В приложениях комментарии обычно объясняют, как работают определенные части программы или какие данные содержатся в какой-либо переменной.

В C# поддерживаются три типа комментариев. Первый, показанный в самом начале программы, называется *многострочным комментарием* (то есть может состоять из нескольких строк). Такой тип комментария должен находиться между символами `/*` и `*/`. Весь текст, находящийся между этими символами, игнорируется компилятором.

Следующая строка программы

```
using System;
```

указывает, что программа использует *пространство имен* `System`. Подробнее о пространстве имен мы расскажем далее, а сейчас коротко заметим, что пространство имен обеспечивает способ хранения одного набора имен отдельно от другого. По существу, имена, объявленные в одном пространстве имен, не конфликтуют с такими же именами, объявленными в другом пространстве имен. Итак, данная программа использует пространство имен `System`, зарезервированное пространством имен для

элементов, которые ассоциированы с библиотекой классов .NET Framework, используемых C#. Ключевое слово `using` означает, что программа использует имена в заданном пространстве имен.

Рассмотрим следующую строку программы:

```
class Example {
```

В этой строке ключевое слово `class` указывает на то, что объявляется новый класс. Уже говорилось, что класс является базовым элементом инкапсуляции в C#. `Example` — это имя класса. Определение класса начинается с открывающей фигурной скобки ({) и заканчивается закрывающей фигурной скобкой (}). Элементы, находящиеся между двумя скобками, называются членами класса. На этом этапе не слишком обращайте внимание на детали класса, учтите только, что в C# все процессы программы происходят в пределах класса. Это один из признаков объектно-ориентированных программ, какими в той или иной мере являются все C#-программы.

Далее следует *однострочный комментарий:*

```
// C#-программа начинается с вызова метода Main().
```

Это второй тип комментариев, которые поддерживаются C#. Он начинается с символа // и заканчивается с окончанием строки. Принято общее правило: многострочные комментарии программист использует для больших заметок, а однострочные — для коротких объяснений, не требующих много места.

Следующая строка кода начинает определение метода `Main ()`:

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

Как уже отмечалось, подпрограмма в C# называется методом. Все приложения на C# начинают выполняться с вызова метода `Main()`. (Так же как программы на C/C++ начинают выполнение с функции `main()`.) Полное объяснение роли каждой части этой строки мы сейчас дать не сможем, поскольку для этого необходимо понимание тонкостей нескольких других свойств C#, но коротко рассмотрим эту строку, поскольку она будет использоваться в большинстве программ данной книги.

Ключевое слово `public` является *модификатором* (или *спецификатором доступа).* Модификатор определяет, как другие части программы могут осуществлять доступ к члену класса. Если перед членом класса находится ключевое слово `public`, доступ к этому члену имеет код, определенный вне класса, в котором объявлен данный член. (Назначение, противоположное модификатору `public`, имеет модификатор `private`, ограждающий член класса от прямого использования кодом, который определен вне его класса.) В этой программе метод `Main()` объявлен как `public`, поскольку при старте программы он будет вызван кодом, определенным вне его класса (в данном случае непосредственно операционной системой).

Примечание

В то время, когда писалась эта книга, в C# фактически не требовалось объявлять метод `Main()` как `public`, но он объявлен таким способом во многих примерах, приведенных в Visual Studio.NET. Кроме того, так предпочитают действовать многие C#-программисты. Поэтому в данной книге мы использовали этот способ, но не удивляйтесь, если в других книгах вам встретится иное объявление метода `Main()`.

Ключевое слово `static` позволяет вызывать метод `Main()` до того, как будет создан объект его класса, а поскольку метод `Main()` вызывается при старте программы, то он обязательно должен быть статическим. Следующее ключевое слово `void` просто сообщает компилятору, что метод `Main()` не возвращает значение. (Далее вы увидите, что методы могут также возвращать значения.) Пустые круглые скобки, которые стоят после слова `Main`, указывают, что методу `Main()` не передается никакая информация. (Далее вы увидите, что существует возможность передавать информацию методу `Main()` или любому другому методу.) Последний символ в строке, символ открывающей фигурной скобки (`{`), указывает начало кода метода `Main()`. Весь код, составляющий метод, помещается между открывающей и закрывающей фигурными скобками.

Приводимый ниже оператор находится внутри метода `Main()`.

```
Console.WriteLine("Простая C#-программа.");
```

Данный оператор выводит строку `"Простая C#-программа."`, затем отображается новая пустая строка. Фактически вывод выполняется встроенным методом `WriteLine()`, а информация, которая передается методу (в данном случае это строка `"Простая C#-программа."`), называется *аргументом*. (В дополнение к строкам метод `WriteLine()` также может использовать другие типы информации для вывода на экран, которые будут рассмотрены далее.) Строка начинается со слова `Console`, которое является именем встроенного класса, поддерживающего выполнение операции ввода/вывода данных. Записывая имя класса `Console` и имя метода `WriteLine()` через точку, вы сообщаете компилятору, что метод `WriteLine()` является членом класса `Console`. Использование класса для определения ввода с клавиатуры является еще одним признаком объектной ориентированности языка C#.

Отметим, что оператор `WriteLine();` заканчивается символом точки с запятой, как и оператор `using System;`, указанный ранее в программе. В C# все операторы заканчиваются этим символом. В нашей программе строки, не заканчивающиеся символом точки с запятой, технически не являются операторами.

Первая закрывающая фигурная скобка в программе завершает метод `Main()`, а последняя завершает определение класса `Example`.

Ответы профессионала

Вопрос. Вы говорили, что C# поддерживает три типа комментариев, но упомянули только два. Какой же третий тип?

Ответ. Третьим типом комментариев в C# является *XML-комментарий*. В нем используются теги XML для поддержки возможности создания самодокументированного кода.

Учтите еще, что в языке C# различаются символы нижнего и верхнего регистров, поэтому ошибка в регистре символа может привести к серьезным проблемам. Например, при вводе `main` вместо `Main` или `writeline` вместо `WriteLine` программа не будет работать. Более того, хотя классы, не содержащие метод `Main()`, скомпилируются, они будут выполнены после их инициализации в другом классе, содержащем этот метод. То есть если вы введете слово `main` с маленькой буквы, компилятор все равно скомпилирует вашу программу, но будет выведено сообщение об ошибке, в котором говорится, что программа `Example.exe` не имеет определенной точки входа.

Минутный практикум

1. С какого метода начинается выполнение C#-программы?
2. Какие действия выполняет метод `Console.WriteLrLne()`?
3. Как называется программа-компилятор, запускаемая из командной строки?

Обработка синтаксических ошибок

Если вы еще этого не сделали, то введите, скомпилируйте и запустите рассмотренную программу. Наверное, вам известно, что при вводе кода можно допустить ошибку. Если введены некорректные данные, компилятор, сделан попытку скомпилировать код программы, сообщит о синтаксической ошибке. Компилятор C# пытается понять ваш исходный код в зависимости от наличия открывающих и закрывающих скобок или признака завершения оператора, поэтому сообщение об ошибке не всегда отражает истинную причину этой ошибки. Например, если в рассмотренной программе будет пропущена открывающая фигурная скобка после имени метода `Main()`, то при компилировании предыдущего файла из командной строки будет сгенерирована приведенная ниже последовательность сообщений об ошибках. (Подобные сообщения об ошибках также генерируются при компилировании с использованием интегрированной среды разработки IDE.)

```
Example.cs(12,28): error CS1002: ; expected
Example.cs(13,22): error CS1S19: Invalid token '(' in class, struct, or
interface member declaration
Example.cs(15,1): error CS1022: Type or namespace definition, or end of
file expected
```

Очевидно, что первое сообщение об ошибке совершенно ошибочно, поскольку пропущена фигурная скобка, а не точка с запятой. Следующие два сообщения тоже сбивают с толку.

Все это мы рассказываем для того, чтобы вы поняли, что не стоит полагаться на информацию, изложенную в сообщении об ошибках, поскольку она может быть неверной. Кроме того, необходимо проверить несколько строк, предшествующих той, номер которой указан в сообщении об ошибке, поскольку иногда сообщение об ошибке появляется только через несколько строк после нее.

Небольшое изменение программы

Хотя все программы в этой книге используют оператор

```
using System;
```

на самом деле в начале первой программы в нем нет технической необходимости. В C# всегда можно *полностью определить* имя класса (или метода), указав пространство имен, к которому класс (или метод) принадлежит. Например, строка

```
Console.WriteLine("Простая C#-программа.");
```

---

1. C#-программа начинает выполняться с вызова метода `Main()`.
2. Метод `Console.WriteLine()` выводит информацию на монитор.
3. Программа-компилятор, запускаемая из командной строки, называется `csc.exe`.

может быть переписана следующим образом:

```
System.Console.WriteLine("Простая C#-программа);
```

Следовательно, первый пример можно записать так:

```
// В этой версии программы не используется оператор
// using System;.

class Example {

  // C#-программа начинается с выполнения метода Main().
  public static void Main() {

    // Здесь метод Console.WriteLine() определен полностью.
    System.Console.WriteLine("Простая C#-програыма.");
  }
}
```

Полное определение метода Console.WriteLine().

Поскольку весьма утомительно каждый раз указывать идентификатор `System`, когда используется член этого пространства имен, большинство C#-программистов все свои программы начинают с оператора `using System;`. Но важно понимать, что при необходимости вы можете определить имя класса или член класса явно, указав использование пространства имен, к которому принадлежит данный класс.

Вторая простая программа

Возможно, оператор присваивания является одним из наиболее важных в программировании. Переменная — это именованная ячейка памяти, которой присваивается значение. И это значение может быть изменено на протяжении выполнения программы. То есть содержимое переменной является изменяемым, а не фиксированным.

В следующей программе создаются две переменные, `x` и `y`:

```
// Эта программа демонстрирует использование переменных.

using System;

class Example2 {
  public static void Main() (
    int x; // В этой строке кода объявляется переменная,
    int y; // В этой строке кода объявляется еще одна переменная.

    x — 100; //   В этой строке кода переменной x
             // присваивается значение 100.


    Console.WriteLine("Переменная x содержит значение " + x);

    y = x / 2;

    Console.WriteLine("Значение переменной y вычисляется");
    Console.Write("с помощью выражения x/2 и равно ");
    Console.WriteLine(y);
  }
}
```

Объявление переменных.

В этой строке кода переменной x присваивается значение 100.

Выполнив программу, вы увидите на экране монитора следующее сообщение:

```
Переменная x содержит значение 100.
Значение переменной y вычисляется
с помощью выражения x/2 и равно 50.
```

В этой программе представлены новые элементы языка C#. Так, оператор

```
int x; // В этой строке кода объявляется переменная.
```

объявляет переменную целочисленного типа с именем `x`. В C# все переменные должны быть объявлены перед их использованием. *Тип* значений, которые может хранить переменная должен быть специфицирован. Принято говорить, что указывается *тип переменной.* В приведенной выше программе переменная `x` может хранить целочисленные значения. В C# для объявления переменной целочисленного типа перед именем переменной необходимо поместить ключевое слово `int`. Следовательно, оператор `int x;-` объявляет переменную с именем `x,` имеющую тип `int`.

В следующей строке кода объявляется вторая переменная, `y`:

```
int y; // В этой строке кода объявляется еще одна переменная.
```

Здесь используется тот же формат объявления, который применялся для переменной `x`, но имя переменной другое.

В большинстве случаев для объявления переменной используется оператор с таким синтаксисом:

```
type var-name;
```

Здесь слово `type` указывает тип объявляемой переменной, а `var-name` — имя переменной. Кроме типа `int` в C# поддерживается несколько других типов данных.

В следующей строке кода переменной `x` присваивается значение 100:

```
x = 100;  // В этой строке кода переменной x присваивается значение 100.
```

В C# оператор присваивания обозначается символом знака равенства. Он копирует значение, находящееся справа от него, в переменную слева от него.

Следующая строка кода выводит значение переменной `x`, перед которым помещается строка "`Переменная x содержит значение `".

```
Console.WriteLine("Переменная x содержит значение " + x);
```

При выполнении этой строки кода в результате применения оператора + значение переменной x будет выведено на экран после строки "`Переменная x содержит значение `". Это значит, что, используя оператор +, можно в пределах одного оператора `WriteLine();` связать вместе сколько угодно элементов. В следующей строке кода переменной `y` присваивается значение переменной `x`, деленное на 2.

```
y = x/2;
```

То есть значение переменной `x` делится на 2 и результат помешается в переменную y. Таким образом, после выполнения оператора переменная y будет иметь значение 50. Значение переменной `x` останется неизменным. Как и многие другие языки программирования, C# поддерживает полный набор арифметических операторов:

```
+     Сложение
-     Вычитание
*     Умножение
/     Деление
```

Рассмотрим следующие три строки кода программы:

```
Console.WriteLine("Значение переменной y вычисляется ");
Console.Write("с помощью выражения x/2 и равно ");
Console.WriteLine(y);
```

Здесь встречаются новый элемент языка и новый вариант использования переменной.

Новый элемент — встроенный метод `Write()`, который используется для вывода на экран строки `"с помощью выражения x/2 и равно"`. За этой строкой *не следует* новая строка. Эго означает, что когда будет сгенерирован следующий вывод, он будет помещен в конец этой же строки. Метод `Write()` похож на метод `WriteLine()`, за исключением того, что он не выводит новую строку после каждого вызова.

Новый вариант применения переменной — самостоятельное использование переменной `y` в вызове метода `WriteLine()`. В C# для вывода значений любых встроенных типов может использоваться как метод `Write()`, так и метод `WriteLine()`.

Теперь рассмотрим еще одну возможность объявления переменных. В C# можно объявлять две переменные и более, используя один и тот же оператор объявления. Для этого нужно просто разделить имена переменных запятыми. Например, переменные `x` и `y` можно объявить таким образом:

```
int x, y; // Для объявления обеих переменных
          // используется один оператор.
```

Другие типы данных

В предыдущей программе была использована переменная типа `int`, которая может хранить только целочисленные значения и не может быть использована для хранения числа с дробной частью. Ранее мы уже говорили, что кроме типа `int` C# поддерживает несколько других типов данных. Для чисел с дробной частью в C# определены два типа, `float` и `double`, которые представляют числа с обычной и двойной точностью соответственно. Чаще всего используется тип `double`.

Для объявления переменных типа `double` используется оператор, синтаксис которого представлен ниже:

```
double result;
```

Здесь слово `result` — это имя переменной, имеющей тип `double`, позволяющий работать с числами с плавающей точкой (например, она может хранить значение 122.23, 0.034, или -19.0).

Чтобы лучше понять различие между типами `int` и `double`, выполните следующую программу:

```
/*
  В этой программе демонстрируется различие между типами.
  int и double.
*/

using System;

class Example3 {
  public static void Main() {
```

```
    inn ivar;     // В этой строке кода объявляется переменная типа int.
    double dvar; // В этой строке кода объявляется переменная типа double,
    ivar = 100;    // Переменной ivar присваивается значение 100.

    dvar = 100.0;  // Переменной dvar присваивается значение 100.0.

    Console.WriteLine("Первоначальное значение переменной ivar: " + ivar);
    Console.WriteLine("Первоначальное значение переменной dvar: " + dvar);

    Console.WriteLine();  // Выводит пустую строку.

    // Значения обеих переменных делятся на 3.
    ivar = ivar / 3;
    dvar = dvar / 3.0;

    Console.WriteLine("Значение переменной ivar после деления:   " + ivar);
    Console.WriteLine("Значение переменной dvar после деления:   " + dvar);
   }
}
```

Вывод пустой строки

Ниже показан результат работы этой программы:

```
Первоначальное значение переменной ivar: 100
Первоначальное значение переменной dvar: 100.0

Значение переменной ivar после деления: 33
Значение переменной dvar после деления: 33.3333333333333
```

Как видите, после деления значения переменной `ivar` на 3 результатом является целое число 33, то есть дробная часть утрачена. А при делении значения переменной `dvar` на 3 дробная часть результата сохраняется. Как показано в программе, при необходимости ввода значения с плавающей точкой, она должна быть указана. В противном случае значение будет интерпретировано как целочисленное. Например, в C# значение 100 является целочисленным, а 100.0 — значением с плавающей точкой.

Заметьте также, что для вывода на экран пустой строки нужно вызывать метод `WriteLine()` без аргументов.

Ответы профессионала

Вопрос. Почему в C# для целочисленных значений и значений с плавающей точкой существуют различные типы данных?

Ответ. C# поддерживает различные типы данных для повышения эффективности программ. Например, операции над целочисленными значениями производятся быстрее, чем над числами с плавающей точкой. Поэтому, если вам не нужны Дробные значения чисел, то нет необходимости производить вычисления с излишней степенью точности, то есть работать с такими типами данных, как `float` и `double`. Кроме того, размер памяти, необходимый для хранения одних типов данных, может быть меньшим, чем размер памяти для хранения других. Предусматривая различные типы данных, C# позволяет эффективно использовать системные ресурсы. Учтите также, что некоторые алгоритмы требуют использования специфических типов данных (по крайней мере, работают при этом более эффективно).

Проект 1-1. Преобразование значений температуры

FtoC.cs

Хотя предыдущие примеры программ и демонстрируют некоторые важные свойства языка C#, на практике такие программы не слишком полезны. Несмотря на то, что на данном этапе ваши знания языка C# довольно скудные, вы все же можете применить их на практике. В этом проекте мы создадим программу, преобразующую значение температуры по шкале Фаренгейта в значение по шкале Цельсия.

В программе объявляются две переменные типа `double`. В одной из них будет храниться значение температуры по шкале Фаренгейта, а в другой — значение по шкале Цельсия, полученное после преобразования. Возможно, вы знаете, что для выполнения этого преобразования необходима следующая формула:

C = 5/9 * (F - 32)

где C — значение температуры по шкале Цельсия (в градусах), а F — значение температуры по шкале Фаренгейта (в градусах).

Пошаговая инструкция

1. Создайте новый C#-файл с именем `FtoC.cs`. (Если компиляция программы производится не из командной строки, а с помощью Visual C++ IDE, необходимо добавить этот файл к C#-проекту, как было описано ранее.)
2. Введите следующую программу в файл:

```
/*
   Проект 1-1

   Эта программа выполняет преобразование значения температуры по шкале
   Фаренгейта и значение температуры по шкале Цельсия.

   Назовите файл FtoC.cs.
*/

using System;

class FtoC {
  public static void Main() {
    double f; // Содержит значение температуры по шкале Фаренгейта,
    double c; // Содержит значение температуры по шкале Цельсия.

   f = 59.0; // Переменная f получает значение 59
             // (градусов по Фаренгейту).

              // Далее выполняется преобразование имеющегося значения
              // в значение температуры по шкале Цельсия.
    c = 5.0 / 9.0 * (f - 32.0);

    Console.Write(f + " градусов по шкале Фаренгейта равны ");
    Console-WriteLine(c + ” градусам по шкале Цельсия.”);
  }
}
```

3. Скомпилируйте программу, используя Visual C++ IDE (следуя ранее данным инструкциям), или введите в командной строке следующую команду:

```
C>csc FtoC.cs
```

4. Запустите программу из Visual C++ IDE или из командной строки, для чего введите после приглашения имя главного класса (содержащего метод `main`):

```
C>FtoC
```

В результате работы программы будет выведена следующая строка:

```
59 градусов по шкале Фаренгейта равны 15 градусам по шкале Цельсия.
```

5. Программа преобразует единственное значение температуры по шкале Фаренгейта (в градусах) в значение температуры по шкале Цельсия. Изменяя число, присваиваемое переменной `f`, вы можете преобразовать любое значение температуры.

Минутный практикум

1. Какое ключевое слово в C# используется для целочисленного типа данных?
2. Что обозначает термин `double`?
3. Обязательно ли в C#-программе использовать оператор `using System;`?

Два управляющих оператора

Внутри метода происходит последовательное выполнение операторов по направлению сверху вниз (то есть так, как они были записаны в данном блоке кода). Однако можно менять этот порядок выполнения с помощью различных управляющих операторов, поддерживаемых C#. Хотя детально управляющие операторы будут рассмотрены в книге позже, о двух из них мы расскажем в этом разделе, поскольку они используются в дальнейших программах.

Оператор `if`

Вы можете избирательно выполнить часть программы с помощью условного оператора `if`. Этот оператор во многом сходен с оператором `if` любого другого языка. В частности, он синтаксически идентичен оператору `if` в языках C, C++ и Java. Самая простая его форма представлена ниже:

```
if (условие) оператор;
```

Здесь *условие* — это булево выражение (имеющее значение `true` или `false`). Если *условие* истинно, то оператор выполняется, если *условие* ложно, то оператор пропускается. Приведем пример использования условного оператора `if`:

```
if (10 < 11) Console.WjriteLine("10 меньше, чем 11");
```

В этом случае число 10 меньше, чем число 11 (условное выражение истинно), следовательно, оператор `WriteLine();` будет выполняться. Рассмотрим пример:

```
if(10 < 9) Console.WriteLine("Эта строка не будет выведена на экран");
```

---

1. Ключевым словом для целочисленного типа данных в C# является слово `int`.
2. `double` — это ключевое слово для данных с плавающей точкой двойной точности.
3. Нет, но это удобно.

В этом случае оператор `WriteLine();` вызываться не будет, поскольку число 10 больше числа 9. В C# определен комплект операторов сравнения, которые могут использоваться в условных выражениях. Ниже представлены эти операторы и их значения.

Оператор	Описание
<	**Меньше чем**
<=	**Меньше или равно**
>	**Больше чем**
>=	**Больше или равно**
==	**Равно**
!=	**Не равно**

Отметим, что оператор равенства состоит из двух символов знака равенства.

Ниже представлена программа, в которой используется оператор `if`:

```
// В программе демонстрируется использование оператора if.

using System;

class IfDemo {
  public static void Main() {
    int a, b, c;

    a = 2;
    b = 3;

    if(a < b) Console.WriteLine("Значение переменной a меньше, чем"+
              " значение переменной b.");

    // Строка, которая приведена в качестве параметра
    // в следующем методе WriteLine(), не будет выведена на экран,
    if(a == b) Console.WriteLine("Эта стоока не будет выведена на экран.");

    Console.WriteLine();

    c = a - b; // Переменной c присваивается значение -1.

    Console.WriteLine("Переменная c содержит значение -1.");
    if(c >= 0) Console.WriteLine("Значение переменной c - " +
               "не отрицательное число.");
    if (c < 0) Console.WriteLine("Значение переменной c — " +
               " отрицательное число.");

    Console.WriteLine();

    c — b - a; // Переменной c присваивается значение 1.
    Console.WriteLine("Переменная c содержит значение 1.");
    if (c >= 0) Console.WriteLine("Значение переменной c - " +
                "не отрицательное число.");
    if (c < 0) Console.WriteLine ("Значение переменной c — " +
                "отрицательное число.");
```

Использование оператора `if`.

При выполнении этой программы на экран были выведены следующие строки:

```
Значение переменной a меньше, чем значение переменной b.

Переменная c содержит значение -1.
Значение переменной c — отрицательное число.

Переменная c содержит значение 1.
Значение переменной c - не отрицательное число.
```

Обратите внимание, что в строке этой программы

```
int a, b, c;
```

объявляются три переменные, которые разделены запятыми. Уже говорилось, что если требуются две или более переменных одного типа, они могут быть объявлены в одном операторе. Для этого нужно разделить имена переменных запятыми.

Цикл `for`

Вы можете многократно выполнять какую-либо последовательность кода (блок), определив *цикл*. В этом разделе мы рассмотрим цикл `for`. В C# этот цикл работает так же, как и в C, C++ и Java. Самый простой синтаксис цикла `for`:

`for (`*`инициализация, условие, итерация`*`) оператор;`

В общепринятой форме часть цикла, отвечающая за его *инициализацию*, присваивает переменной цикла некоторое начальное значение. *Условие* является булевым выражением, проверяющим переменную цикла. Если результатом этого теста является значение true, цикл for выполняется далее, если — false, происходит естественное завершение работы цикла. *Итерационное* выражение определяет, как изменяется контрольная переменная цикла при каждом его выполнении. Далее представлена короткая программа, в которой демонстрируется использование цикла for:

```
// Программа, в которой демонстрируется использование цикла for.

using System;

class ForDemo {
  public static void Main() {
    int count;

    for(count = 0; count < 5; count = count+1) {
      int num = count +1;
      Console.Writeline("Это " + num + "-й проход цикла.");
    )
    Console.Writeline("Цикл завершен.");
  }
}
```

Ниже показан результат выполнения этой программы:

```
Это 1-й проход цикла.
Это 2-й проход цикла.
Это 3-й проход цикла.
Это 4-й проход цикла.
Это 5-й проход цикла.
Цикл завершен.
```

В этом примере `count` является переменной цикла. Первоначально в части инициализации цикла `for` ей задается значение ноль. В начале каждого повторения цикла (включая первое) выполняется проверка условия `count < 5`. Если условие истинно, то выполняется оператор `WriteLine()`, а затем инициализируется итерационная часть цикла (то есть переменной `count` присваивается новое значение, равное `count + 1`). Этот процесс повторяется до тех пор, пока значение переменной цикла перестанет удовлетворять условию. С этого момента выполнение никла прекращается. Переменная `num`, была введена в программу только для того, чтобы в выводимых строках нумерация проходов цикла начиналась не с ноля, а с единицы.

В профессионально написанной C#-программе вы практически не встретите итерационную часть цикла, написанную так, как в предыдущем примере. То есть операторы, подобные приведенному ниже, встречаются в цикле `for` крайне редко.

```
count = count + 1;
```

Причина этого в том, что в C# есть специальный оператор инкремента, который выполняет операцию увеличения первоначального значения переменной на единицу более эффективно. Это оператор, обозначаемый двойным символом знака плюс (++). Итерационная часть приведенного выше цикла `for` с использованием оператора инкремента записывается следующим образом:

```
count++;
```

А весь цикл `for` будет выглядеть так:

```
for(count - 0; count < 5; count++)
```

Работать он будет точно так же, как предыдущий.

Также в C# имеется оператор декремента, обозначаемый двойным символом знака минус (`--`). Этот оператор уменьшает свой операнд на единицу.

Минутный практикум

1. Как звучит полное название оператора `if`?
2. К каким операторам относится `for`?
3. Какие операторы сравнения имеются в C#?

Использование блоков кода

Еще одним ключевым элементом языка C# является *блок кода.* Блок кода — это объединение двух или более операторов, заключенное в фигурные скобки. После создания блок кода становится логическим элементом, который можно использовать так же, как одиночный оператор. Блок может использоваться в операторах `if` и `for`. Рассмотрим следующий фрагмент кода:

```
if (w < h) {
   v = w * h;
   w = 0;
}
```

1. `if` — это условный оператор.
2. `for` относится к операторам цикла.
3. В C# имеются следующие операторы сравнения: <, <=, >, >=, ==, !=.

Если в данном операторе `if` значение переменной `w` меньше значения переменной `h`, то оба оператора внутри блока будут выполнены. То есть поскольку два оператора внутри блока формируют логическую единицу, один оператор не может выполняться без выполнения другого. Это означает, что если вам понадобится логическая связь между двумя операторами, вы должны поместить их в блок. Блоки кода позволяют реализовать многие алгоритмы так, чтобы они были понятны и выполнялись с высокой эффективностью. Ниже представлена программа, в которой блок кода используется для исключения возможности деления на ноль:

```
// В программе демонстрируется использование блока кода.

using System;

class BlockDemo {
  public static void Main() {
    int i, j, d;

    i = 5;
    j = 10;

    // Этот блок кода относится к оператору if,
    if (j != 0){
      Console.WriteLine("Значение переменной i не равно нолю.");
      d = j / i;
      Console.WriteLine("j/i равно " + d);
    }
  }
}
```

Все это блок кода относится к оператору if

Ответы профессионала

Вопрос. Влияет ли использование блока кода на эффективность программы (то есть не увеличивается ли время ее выполнения)?

Ответ. Использование блоков кода не увеличивает время выполнения программы. Поскольку они способны упростить кодирование определенных алгоритмов, их использование, как правило, увеличивает скорость работы программы и ее эффективность.

Результат выполнения этой программы представлен ниже:

```
Значение переменной i не равно нолю.
j/i равно 2
```

В данной программе к оператору `if` относится блок кода, а не одиночный оператор. Если условие, управляющее оператором `if`, истинно (как в данном случае), то выполняются все три оператора в блоке. Попробуйте присвоить переменной `i` значение `0` и посмотрите, каков будет результат.

Как вы увидите далее, блоки кода имеют дополнительные свойства и способы применения, но основное их предназначение — создание логически неразделимой единицы кода.

Символ точки с запятой и позиционирование

В C# символ точки с запятой означает окончание оператора. То есть каждый отдельный оператор должен заканчиваться точкой с запятой.

Как вы знаете, блок является набором логически соединенных операторов, заключенных в фигурные скобки, поэтому его завершает не точка с запятой, а закрывающая фигурная скобка.

C# не распознает окончание строки как окончание оператора, только точка с запятой может завершить оператор. Поэтому не имеет значения, в каком месте строки вы помещаете точку с запятой. Например, для C# строки кода

```
x = y;
y - y + 1;
Console.WriteLme(x + " " + y);
```

будут означать то же, что и строка

```
x = y; y = y + 1; Console.WriteLine(x + y);
```

Более того, отдельные элементы оператора также могут быть помещены в разные строки. Так, допустимо следующее расположение:

```
Console.WriteLine("Эта строка при выводе займет довольно много \n" +
          "места, потому что кроме  данных " + x + y + z +
          "\n она состоит из трех  строковых литералов.");
```

Такой способ разбивки длинных строк используется для улучшения читабельности программ. К тому же можно увидеть, как эта строка будет выглядеть при выводе.

Использование отступов

Вероятно, вы заметили, что в предыдущих примерах некоторые операторы написаны с отступом. C# является языком, в котором применение различных отступов или их отсутствие при вводе кода (при вводе операторов) не оказывает влияния на работоспособность программы. Но поскольку существует общепринятый стиль написания программ, которому следует данная книга, советуем вам его придерживаться. Для этого нужно при вводе кода делать отступ (два-три пробела относительно начала верхней строки) после каждой открывающей скобки и возвращаться обратно после каждой закрывающей скобки. Исключение составляют операторы, для которых предусмотрен дополнительный отступ; они будут перечислены позже.

Минутный практикум

1. Как создать блок кода?
2. Как завершаются операторы в C#?
3. Справедливо ли утверждение, что все C#-операторы должны начинаться и заканчиваться на одной строке?

---

1. Последовательность операторов помещается между открывающей ({) и закрывающей (}) фигурными скобками. Блок создает логическую единицу кода.

2. Операторы в C# завершаются символом точки с запятой.

3. Нет.

Проект 1-2. Усовершенствование программы по преобразованию значения температуры

FtoCTable.cs

Используя цикл `for` и оператор `if`, мы усовершенствуем разработанную в первом проекте программу, преобразующую значение температуры по шкале Фаренгейта в значение по шкале Цельсия. В новой версии на экран выводится таблица преобразованных значений, начиная с 0 градусов по шкале Фаренгейта и заканчивая 99 градусами. После вывода десяти строк с исходными и преобразованными данными выводится пустая строка. Это выполняется с помощью переменной `counter`, используемой для подсчета выведенных строк (такой алгоритм называется *использованием счетчика).*

Пошаговая инструкция

1. Создайте новый файл с именем `FtoCTable.cs`.
2. Введите следующую программу:

```
/*
   Проект 1-2

   В этой программе выводится таблица значений температуры
   по шкале Фаренгейта и соответствующих им значений
   температуры по шкале Цельсия.

   Сохраните эту программу в файле FtoCTable.es.
*/

using System;

class FtoCTable {
  public static void MainO {
    double f, c;
    int counter;

    counter  =0;
    for(f = 0.0; f < 100.0; f++)  (
      // Преобразование в градусы по шкале Цельсия,
      c « 5.0 / 9.0 * (f - 32.0);
      Console-WriteLine (f + "( по шкале Фаренгейта равны " <-
                         c f "( по шкале Цельсия.");

      counter++;

      // После каждых десяти строк выводится пустая строка.
      if(counter == 10){
        Console.WriteLine();
        counter = 0; // Для отсчета очередных 10 строк.
      )
    )
  }
}
```

Счетчику строк присваивается первоначальное значение 0.

Увеличение значения счетчика строк на 1 при каждом прохождении цикла.

Если значение переменной counter равно 10, выводится нусгая строка.

3. Скомпилируйте программу в Visual C++ IDE или введите в командной строке следующую команду:

```
C>csc FtoCTable.cs
```

4. Запустите программу, используя Visual C++ IDE, или введите в командной строке имя программы:

```
C>FtoCTable
```

Ниже представлена часть результата работы программы `FtoCTable`.

```
0° по шкале Фаренгейта   равны -17.7777777777778° по шкале Цельсия.
1° по шкале Фаренгейта   равны -17.2222222222222° по шкале Цельсия.
2° по шкале Фаренгейта   равны -16.6666666666667° по шкале Цельсия.
3° по шкале Фаренгейта   равны -16.1111111111111° по шкале Цельсия.
4° по шкале Фаренгейта   равны -15.5555555555556° по шкале Цельсия.
5° по шкале Фаренгейта   равны -15° по шкале Цельсия.
6° по шкале Фаренгейта   равны -14.4444444444444° по шкале Цельсия.
7° по шкале Фаренгейта   равны -13.8888888888889° по шкале Цельсия.
8° по шкале Фаренгейта   равны -13.3333333333333° по шкале Цельсия.
9° по шкале Фаренгейта   равны -12.7777777777778° по шкале Цельсия.

10° по шкале Фаренгейта  равны -12.222222222222° по шкале Цельсия.
11° по шкале Фаренгейта  равны -11.666666666667° по шкале Цельсия.
12° по шкале Фаренгейта  равны -11.111111111111° по шкале Цельсия.
13° по шкале Фаренгейта  равны -10.555555555556° по шкале Цельсия.
14° по шкале Фаренгейта равны -10° по шкале Цельсия.
15° по шкале Фаренгейта  равны -9.4444444444444° по шкале Цельсия.
16° по шкале Фаренгейта  равны -8.8888888888889° по шкале Цельсия.
17° по шкале Фаренгейта  равны -8.3333333333333° по шкале Цельсия.
18° по шкале Фаренгейта  равны -7.7777777777778° по шкале Цельсия.
19° по шкале Фаренгейта  равны -7.2222222222222° по шкале Цельсия.
```

Ключевые слова в языке C#

В языке C# определены 77 ключевых слов (табл. 1.1), которые нельзя использовать в качестве имен для переменных, классов и методов.

Таблица 1.1. Ключевые слова C#

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

Идентификаторы

Идентификаторами в C# являются имена методов, переменных или других определенных пользователем элементов. Идентификаторы могут состоять из одного или

нескольких символов. Имена переменных могут начинаться с любой буквы или знака подчеркивания. Вторым символом идентификатора может быть буква, знак подчеркивания или цифра. Знаки подчеркивания используются для улучшения читабельности имени переменной (например, `line_count`). Компилятор различает символы верхнего и нижнего регистров (то есть `myvar` и `MyVar` — разные имена). Ниже приведено несколько примеров допустимых идентификаторов.

```
Test     x         y2       MaxLoad
up       _top      my_var   sample23
```

Запомните, что идентификатор не может начинаться с цифры. Следовательно, 12x. не является действительным именем идентификатора. Хороший стиль программирования предполагает использование имен идентификаторов, которые отражают значение или особенности применения данного элемента.

Хотя в C# в качестве имени идентификатора нельзя использовать ключевое слово, перед ним можно ставить символ @, что позволит использовать ключевое слово в качестве действительного идентификатора. Например, `@for` является действительным идентификатором. Фактически в этом случае идентификатором является `for`, а символ @ будет игнорироваться. Использование ключевых слов с начальным символом @ не рекомендуется, за исключением тех случаев, когда это необходимо для специальных целей. Символ @ также может стоять в начале любого имени идентификатора, но использовать его без особой необходимости не рекомендуется.

Минутный практикум

1. Какое из трех слов является ключевым — `for`, `For` или `FOR`?
2. Какие символы может содержать идентификатор в языке C#?
3. Являются ли идентификаторы index21 и Index2l одинаковыми?

Библиотека классов C#

В программах, представленных в этой главе, были использованы два встроенных метода, `WriteLine()` и `Write()`. Как уже говорилось, эти методы являются членами класса `Console`, принадлежащего пространству имен `System`, которое в свою очередь определено в библиотеке классов. NET Framework. Библиотека классов.NET Framework используется в C# для предоставления поддержки операций ввода/вывода, обработки строк, работы в сети и создания интерфейсов GUI. C# тесно интегрирован со стандартными классами платформы .NET. Как вы увидите, библиотека классов обеспечивает большинство функциональных возможностей, использующихся в любой C#-программе. Чтобы стать программистом на C#, необходимо изучить эти стандартные классы. В этой книге описаны различные методы и элементы библиотеки классов .NET, но поскольку библиотека довольно большая, многие ее компоненты вы должны будете изучить самостоятельно.

---

1. Ключевым является слово `for`. В C# все ключевые слова следует писать только строчными буквами.

2. Идентификатор может содержать буквы, цифры и знаки подчеркивания.

3. Нет, в C# учитывается регистр символов.

Контрольные вопросы

1. Что такое MSIL и почему он так важен для C#?
2. Что такое не зависящая от языка среда исполнения (CLR)?
3. Каковы три основных принципа объектно-ориентированного программирования?
4. С какого метода начинается выполнение программ, написанных на C#?
5. Что такое переменная? Что такое пространство имен?
6. Какое из следующих имен переменных является недействительным в C#?
   а) count
   б) $count
   в) count 27
   г) 67count
   д) @if
7. Как создать однострочный комментарий? Как создать многострочный комментарий?
8. Запишите синтаксис оператора `if`. Запишите синтаксис цикла `for`.
9. Как создать блок кода?
10. Обязательно ли начинать каждую C#-программу следующим оператором:
    ```
    using System;
    ```
11. Сила тяжести на поверхности Луны составляет приблизительно 17 процентов от силы тяжести на поверхности Земли. Напишите программу, которая бы вычисляла, каким будет ваш вес на Луне.
12. Измените проект 1-2 таким образом, чтобы программа выводила таблицу преобразования дюймов в метры. Результат должен вычисляться для множества расстояний, взятых в диапазоне от 0 до 100 дюймов. После каждых 12 строк программа должна выводить пустую строку. (Один метр приблизительно равен 39.37 дюйма.)

Типы данных и операторы

- Базовые типы данных C#
- Литералы и их использование
- Создание инициализированных переменных
- Область видимости переменных
- Арифметические операторы
- Операторы сравнения и логические операторы
- Оператор присваивания
- Операция приведения типа, явное и неявное преобразование типов данных
- Выражения в C#

Типы данных и операторы составляют основу любого языка программирования. Эти элементы определяют возможности языка и категории задач к которым может быть применен этот язык C# поддерживает большое количество типов данных и операторов, что делает его удобным для решения многих задач программирования

Рассмотрение операторов и типов данных требует довольно много времени. В этой главе мы поговорим об основных типах данных и наиболее часто используемых операторах, а также подробно расскажем о переменных и выражениях

Строгий контроль типов данных в C#

C# — язык со строгим контролем типов данных Это означает, что все операции в C# контролируются компилятором на предмет совместимости типов данных, а если операция является недопустимой, то она не будет компилироваться. Такая строгая проверка типов данных помогает предотвратить ошибки и повышает надежность Для возможности осуществления этого контроля всем переменным, результатам вычислении выражении и значениям задан определенный тип (то есть не существует переменной с неопределенным типом) Более того, тип значения определяет виды операции, которые разрешено производить над ним. Операция разрешенная для одного типа данных, может быть запрещена для другого типа

Обычные типы данных

C# включает две основные категории встроенных типов данных: обычные типы (или простые типы) и ссылочные типы. К ссылочным типам относятся классы, о которых мы поговорим позже. Обычные типы, которых в ядре C# тринадцать, представлены в табл. 2.1.

Таблица 2 1 Обычные типы данных

Тип	Описание
`Bool`	Значения `true/false` (истина/ложь)
`Byte`	8 битовое беззнаковое целое
`Char`	Символ
`Decimal`	Числовой тип для финансовых вычислений
`Double`	Число двойной точности с плавающей точкой
`Float`	Число одинарной точности с плавающей точкой
`Int`	Целое число
`Long`	Длинное целое
`Sbyte`	8 битовое знаковое целое
`Short`	Короткое целое
`Uint`	Беззнаковое целое
`Ulong`	Беззнаковое длинное целое
`Ushort`	Беззнаковое короткое целое

В C# для каждого типа данных строго специфицированы диапазон значении и возможные операции над ними Соответственно требованиям переносимости в C#

строго контролируется выполнение этих спецификаций. Например, тип данных int одинаков для любой среды выполнения программ, поэтому нет необходимости переписывать код для обеспечения совместимости этого типа с каждой отдельной платформой. Строгая спецификация необходима для достижения переносимости, хотя на некоторых платформах может привести к небольшой потере производительности.

Целочисленные типы

В C# определены девять целочисленных типов данных: char, byte, sbyte, short, ushort, int, uint, long и ulong. Тип данных char в основном применяется для представления символов и будет описан позже в этой главе. Остальные восемь типов используются для выполнения операций с числами. В следующей таблице перечислены эти типы, а также указано количество битов, выделяемых для представления числа, и диапазон возможных значений.

Тип	Количество битов	Диапазон значений
byte	8	От 0 до 255
sbyte	8	От -128 до 127
short	16	От -32768 до 32767
ushort	16	От 0 до 65535
int	32	От -2147483648 до 2147483647
uint	32	От 0 до 4294967295
long	64	От -9223372036854775808 до 9223372036854775807
ulong	64	От 0 до 18446774073709551615

Как показано в таблице, в C# определены как знаковые, так и беззнаковые варианты целочисленных типов. Различие между ними состоит в способе интерпретации старшего бита целого числа. Если указывается знаковое целое, то компилятор будет генерировать код, в котором предполагается интерпретация старшего бита целого числа как *флага знака.* Если флаг знака 0, то это положительное число; если флаг-знака 1, то число отрицательное. Отрицательные числа практически всегда представлены с использованием метода *двоичного дополнения.* В этом методе вес биты числа (за исключением флага знака) инвертируются, затем к этому числу прибавляется единица, в завершение флагу знака задается значение 1.

Знаковые целые важны для очень многих алгоритмов, но они имеют только половину абсолютной величины соответствующих беззнаковых целых. Например, запишем в Двоичном представлении число типа short 32767:

```
01111111 11111111
```

Поскольку это знаковый тип, при задании старшему биту значения 1 число будет интерпретировано как -1 (если использовать метод двоичного дополнения). Но если вы объявляете его тип как ushort, то при задании старшему биту значения 1 число будет интерпретировано как 65535.

Пожалуй, наиболее часто используемым целочисленным типом является int. Переменные типа int часто применяют для управления циклами, для индексации массивов и в различных математических вычислениях. При работе с целочисленными значениями, большими, чем те, которые могут быть представлены типом int, в C# используют типы uint, long, ulong, а при работе с беззнаковым целым — тип uint.

Для больших значений со знаком применяют тип long, для еще больших положительных чисел (беззнаковых целых) — тип ulong.

Ниже приведена программа, вычисляющая объем куба (в кубических дюймах), длина стороны которого равна 1 миле. Поскольку это значение довольно большое, для его хранения программа использует переменную типа long.

```
/*
  Программа вычисляет объем куба (в кубических дюймах),
  длина стороны которого равна 1 миле.
   (Для справки: в одной миле 5280 футов, в одном футе 12 дюймов.
*/

using System;

class Inches {
  public static void Main() {
    long ci;
    long im;

    im = 5280 * 12;

    ci = im * im * im;

    Console.WnteLine("Объем куба с длиной стороны, равной 1 миле, " +
             "\nравен "+ ci +" кубических дюймов.");

  }
}
```

Ниже показан результат выполнения этой программы:

```
Объем куба с длиной стороны, равной 1 миле,
равен 254358061056000 кубических дюймов.
```

Очевидно, что такой результат не может быть помещен в переменную типа int или uint.

byte и sbyte — наименьшие целочисленные типы. Значение типа byte может находиться в диапазоне от 0 до 255. Переменные типа byte особенно полезны при использовании необработанных двоичных данных, таких как поток бантов данных, сгенерированный каким-либо устройством. Для небольших знаковых целых применяется тип sbyte. Представленная ниже программа использует переменную типа byte для контроля цикла for, в котором вычисляется сумма всех целых чисел, находящихся в диапазоне от 1 до 100.

```
// Использование тика byte.

using System;

class Use_byte {
  public static void Main() {
    byte x;
    int sum;

    sum = 0;
    for(x — 1; x <= 100; x++)
      sum = sum + x;
```

```
    Console.WriteLine("Сумма всех целых чисел, находящихся "+
                      "в диапазоне от 1 до 100, \nравна " + sun);
  }
}
```

Результат выполнения этой программы следующий:

```
Сумма всех целых чисел, находящихся в диапазоне от 1 до 100, равна 5050
```

Поскольку в данной программе для управления циклом `for` используются числа от 1 до 100, находящиеся в диапазоне значений, определенных для типа `byte`, нет необходимости назначать тип переменной, позволяющий работать с гораздо большими числами.

Для правильного назначения типа переменной и экономии системных ресурсов воспользуйтесь приведенной выше таблицей и выберите диапазон значений типа, которому удовлетворяет значение этой переменной.

Типы данных с плавающей точкой

Как уже говорилось в главе 1, типы данных с плавающей точкой могут представлять числа, имеющие дробные части. В C# существуют два типа данных с плавающей точкой, `float` и `double`, представляющие числа с одинарной и двойной точностью соответственно. Для представления данных типа `float` выделяются 32 бита, что позволяет присваивать переменным значения чисел, находящихся в диапазоне от 1.5E-45 до 3.4E+38. Для представления данных типа `double` выделяются 64 бита, что расширяет диапазон используемых чисел до величин из диапазона от 5E-324 до 1.7E+308. Чаще всего применяется тип `double`. Одна из причин этого в том, что многие математические функции в библиотеке классов C# (которой является библиотека .NET Framework) используют значения, имеющие тип `double`. Например, метод `Sqrt()`, определенный в стандартном классе `System.Math`, возвращает значение типа `double`, являющееся квадратным корнем его аргумента, также имеющего тип `double`. Ниже в качестве примера приведена программа, в которой для вычисления длины гипотенузы, заданной длинами двух катетов, используется метод `Sqrt()`.

```
/*
   В программе используется теорема Пифагора, позволяющая найти
   длину гипотенузы по известным длинам катетов.
*/

using System;

class Hypot {
  public static void Main() {
    double x, y, z;

    x = 3;
    y = 4;

    z = Math.Sqrt(x*x + y*y);

    Console.WriteLine("Длина гипотенузы равна " + z);
  }
}
```

Обратите внимание на способ вызова метода `Sqrc()`. Имя метода отделено точкой от имени класса, членом которого он является.

Результат выполнения этой программы:

```
Длина гипотенузы равна 5
```

Отметим еще одну особенность данной программы. Как уже говорилось, метод `Sqrt()` является членом класса `Math`. Обратите внимание на способ вызова метода `Sqrt()` — имя метода отделено точкой от имени класса, членом которого он является. Похожий способ записи нам уже встречался, когда перед именем метода `WriteLine()` стояло имя его класса `Console`. Не все стандартные методы нужно вызывать, указав вначале имя класса, в котором определен данный метод, но некоторые методы требуют именно такого вызова.

Тип `decimal`

Возможно, наиболее интересным числовым типом данных в C# является тип `decimal`, предназначенный для использования в денежных вычислениях. В типе `decimal` для представления значений, находящихся в диапазоне от 1E-28 до 7.9E+28, используется 128 битов. Вы, конечно, знаете, что в обычных арифметических вычислениях, производимых над числами с плавающей точкой, неоднократные округления значений приводят к неточному результату. Тип данных `decimal` устраняет ошибки, возникающие при округлении, и может представлять числа с точностью до 28 десятичных разрядов (а в некоторых случаях и до 29 разрядов). Эта способность представлять десятичные значения без ошибок округления особенно полезна, когда рассчитываются финансы.

В качестве примера рассмотрим программу, которая для денежных вычислений использует тип данных `decimal`. Программа вычисляет баланс после начисления процентов.

```
/*
  Программа демонстрирует использование типа decimal
  для финансовых вычислений.
*/

using System;

class UseDecimal {
  public static void Main() {
    decimal balance;
    decimal rate;

    // Подсчет нового баланса
    balance = 1000.10m;
    rate = 0.1m;
    balance = balance * rate + balance;

    Console.WriteLme("Новый баланс: $" + balance);
  }
}
```

При вводе значения типа decimal должны заканчивайся символом m или M.

Эта программа выводит следующий результат:

```
Новый баланс: 51100.11
```

Ответы профессионала

Вопрос. Ранее я работал с языками программирования, в которых не было типа данных `decimal`. Является ли он уникальным и присущим только C#?

Ответ. Да, этот тип данных уникальный; такими языками, как C, C++ или Java, он не поддерживается.

Отметим, что после десятичных констант (то есть констант, имеющих тип `decimal`) должен следовать суффикс `m` или `M`, иначе эти значения будут интерпретированы как константы с плавающей точкой, не совместимые с типом данных `decimal` (Далее в этой главе мы подробно рассмотрим, как специфицируются числовые константы)

Минутный практикум

1. Какие целочисленные типы данных существуют в C#?
2. Назовите два типа данных с плавающей точкой.
3. Почему тип данных `decimal` так важен для финансовых вычислении?

Символы

В отличие от других языков программирования (таких, как C+ + , в которых для представления символа выделяется 8 битов, что позволяет работать с 255 символами) в C# используется стандартный набор символов Unicode, в котором представлены символы всех языков мира. В C# `char` — беззнаковый тип, которому для представ-ления данных выделено 16 битов, что позволяет работать со значениями, находящимися в диапазоне от 0 до 65535 (то есть с 65535-ю символами). Стандартный 8-битовый набор символов ASCII является составной частью Unicode и находится в диапазоне от 0 до 127. Таким образом, ASCII-символы остаются действительными в C#

Для того чтобы присвоить значение символьной переменной, нужно заключить в одинарные кавычки символ, стоящий справа от оператора присваивания. Ниже показан синтаксис операторов, при выполнении которых переменной `ch` присваивается значение `X`.

```
char ch;
ch = 'x';
```

Для вывода символьного значения используется оператор `WriteLine()`. Следующая строка выводит значение переменной `ch`.

```
Console.WriteLine("Значение переменной ch - " + ch);
```

Тип `char`, определенный в C# как целочисленный тип данных, никогда не может быть свободно совмещен с целочисленными типами, предназначенными для

---

1. В C# определены следующие целочисленные типы данных: `byte`, `short`, `int`, `long`, `sbyte`, `ushort`, `uint` и `ulong`. Тип данных `char` также технически является целочисленным, но в основном он используется для хранения символов.
2. Типы данных с плавающей точкой — `float` и `double`
3. Данные типа `decimal` используются для финансовых вычислений, поскольку они не подвержены ошибкам округления.

представления чисел, поскольку отсутствует автоматическое преобразование из целочисленного типа в `cnar`. Например, следующим фрагмент кода недействителен:

```
char ch;

ch = 10; // Данный оператор недействителен.
```

Причина этого в том, что значение 10 — целочисленное и не будет автоматически конвертировано в тип `char`. Следовательно, операнды данной операции присваивания представляют собой несовместимые типы. Если вы попытаетесь скомпилировать этот код, компилятор сообщит об ошибке.

Далее в этой главе мы расскажем, как можно обойти это ограничение.

Ответы профессионала

Вопрос. Почему в C# используется Unicode?

Ответ. В задачи разработчиков C# входило создание такого компьютерного языка, который позволял бы писать программы, предназначенные для использования во всем мире. Поэтому авторы воспользовались стандартным набором символов Unicode, в котором представлены все языки мира. Конечно, использование этого стандарта неэффективно для таких языков, как английский, немецкий, испанский или французский, символы которых могут быть представлены 8 битами, но это цена, которую приходится платить за переносимость программ в глобальном масштабе.

Булев тип данных

Для булева типа данных, `bool`, в C# определены два значения `true` и `false` (истина и ложь). Следовательно, переменная типа `bool` или логическое выражение будут иметь одно из этих двух значений. Более того, не существует способа преобразования значений типа `bool` в целочисленные значения. Например, значение 1 не преобразуется в значение `true`, а значение 0 — в значение `false`.

Приведем программу, демонстрирующую использование типа данных `bool`.

```
// Программа демонстрирует использование
// переменной, имеющей тип bool.

using System;

class BoolDemo {
  public static void Main() {
    bool b;

    b = false;
    Console.WriteLine("Переменная b имеет значение " + b);
    b = true;
    Console. WriteLine("Переменная b имеет значение " + b) ;

    // Булева переменная может управлять условным оператором if.
    if(b) Console.WnteLine("Этот оператор будет выполнен.");

    b = false;
```

Булева переменная может управлять условным оператором `if`

```
    // Условный оператор возвращает булево значение

  Console.WriteLine("(10 > 9)-это " + (10 > 9));
  }
}
```

Ниже представлен результат, сгенерированным этом программой

```
Переменная b имеет значение False
Переменная b имеет значение True
Этот оператор будет выполнен.

(10 > 9) - это True
```

Обратите внимание на некоторые особенности использования булевой переменной. Во-первых, при выводе значения типа `bool` с помощью метода `WriteLine()` на экран выводится слово `true` или `false`. Во-вторых, используя переменную типа `bool`, можно управлять оператором `if`. Если условием выполнения оператора `if` является истинность переменной (в данной программе переменной `b`), то нет необходимости записывать оператор `if` так:

```
if(b == true)...
```

достаточно более короткого выражения

```
if(b)
```

В-третьих, результатом выполнения оператора сравнения, такого как <, является значение типа `bool`. Вот почему выражение `10 > 9` выводит на экран значение `true`. Далее, дополнительная пара скобок, в которые заключено выражение `10 > 9`, является необходимой, поскольку оператор + имеет больший приоритет, чем оператор >.

Минутный практикум

1. Что такое Unicode?
2. Какие значения может иметь переменная типа `bool`?
3. Каков размер данных типа `char` в битах?

Форматирование вывода

Прежде чем продолжить рассмотрение типов данных и операторов, сделаем небольшое отступление До этого момента при выводе списка данных мы разделяли части списка с помощью знака плюс, как показано ниже:

```
Console.WriteLine("Было заказано " + 2 + " книги по $" + 3 +
                  "за единицу товара.");
```

Это удобно, но такой вывод числовой информации не позволяет контролировать способ появления данной информации на экране. Например, нельзя контролировать выводимое число десятичных разрядов для значений с плавающей точкой. Рассмотрим следующий оператор:

```
Console.WriteLine("10/3 = " + 10.0/3.0);
```

1. Unicode — это набор международных символов, каждый из которых представлен 16 битами
2. Переменная типа `bool` может иметь значение либо `true`, либо `false`
3. Данные типа `char` представлены 16 битами

После его выполнения на экран будет выведена следующая строка:

```
10/3 = 3.33333333333333
```

Для некоторых целей вывод такого большого числа десятичных разрядов оправдан, но в других случаях он может быть неприемлем. Например, в финансовых вычислениях обычно требуется вывод только двух десятичных разрядов.

Для форматирования числовых данных необходимо использовать перегруженный метод `WriteLine()` (перегрузка методов будет рассматриваться в главе 6), в один из параметров которого можно внедрить форматирующую информацию. Синтаксис этого метода выглядит так:

```
WriteLine("форматирующая строка", arg0, arg1, ..., argN);
```

В этой версии метода `WnteLine()` аргументы разделены запятыми, а не знаками плюс. Форматирующая строка состоит из стандартных выводимых без изменения символов и спецификаторов формата. Синтаксис спецификаторов формата обычно выглядит так:

```
{argnum, width: fmt}
```

Здесь элемент `argnum` указывает номер аргумента (начиная с ноля), который должен быть выведен на экран в данном месте строки. Элемент `width` определяет минимальную ширину поля, предоставляемого данному аргументу, а элемент `fmt` специфицирует используемый формат вывода.

Когда при выполнении в форматирующей строке встречается спецификатор формата, программа заменяет его соответствующим аргументом, который и выводится на экран. Следовательно, позиция спецификатора формата в форматирующей строке указывает место, где на экране будут выведены соответствующие данные. Элементы `width` и `fmt` являются не обязательными. Следовательно, в своей простейшей форме спецификатор формата указывает аргумент, необходимый для вывода на экран. Например, спецификатор `{0}` указывает на аргумент `arg0`, спецификатор `{1}` указывает на аргумент `arg1` и так далее. Теперь рассмотрим простой пример. Оператор

```
Console.WriteLine("В феврале {0} или {1} дней.", 28, 29);
```

выведет на экран следующую строку:

```
В феврале 28 или 29 дней.
```

Как видите, спецификатор `{0}` заменяется значением `28`, а спецификатор `{1}` - значением `29`. Обратите внимание, что аргументы метода `WriteLine()` разделяются запятыми, а не знаками плюс.

Следующий пример демонстрирует усовершенствованную версию предыдущего оператора, в которой специфицирована минимальная ширина поля:

```
Console.WriteLine("В феврале {0,10} или {1,5} дней.", 28, 29);
```

Результат выполнения данного оператора следующий:

```
В феврале 28 или 29 дней.
```

Если выводимое число меньше заданной ширины поля, то неиспользованные части поля заполняются пробелами. Помните, что ширина поля — это минимальное количество позиций, выделенных для выводимого значения. Если для вывода числа потребуется большее количество позиций, они будут выделены автоматически.

В предыдущих примерах к самим значениям форматирование не применялось, по безусловно, используемые спецификаторы формата должны контролировать отображение данных. Чаще всего форматируются значения с плавающей точкой и значения, имеющие тип `decimal`. Одним из простейших способов спецификации формата является написание шаблона, который будет использован методом `WriteLine()` при форматировании выводимых значений. Шаблон создается следующим образом: в спецификаторе формата указывается номер аргумента, затем после двоеточия при помощи символа `#` указываются позиции выводимых цифр. Так, после выполнения оператора

```
Console.WriteLine("10/3 - {0:#.##}", 10.0/3.0);
```

программа выведет отформатированное значение, являющееся результатом деления 10.0 на 3.0,

```
10/3 = 3.33
```

В этом примере шаблон создается последовательностью символов `#.##`, с помощью которых методу `WriteLine()` сообщается о необходимости вывода двух десятичных разрядов. Если программой выводится число, у которого целая часть содержит более одного знака, то во избежание потери данных метод `WriteLine()` будет выводить более одной цифры слева от запятой.

В случае необходимости вывода значения в долларах и центах используйте спецификатор формата `C`. Например,

```
decimal balance

balance = 12323.09m;
Concole.WriteLine("Текущий баланс равен: {0:C}", balance);
```

В результате будет выведено следующее сообщение:

```
Текущий баланс равен: $12,323.09
```

Проект 2-1. Разговор с Марсом

`Mars.cs`

Наименьшее расстояние между Марсом и Землей составляет примерно 34 миллиона миль. Предположим, что на Марсе находится человек, с которым нам необходимо поговорить. Для этого нужно написать программу, которая могла бы вычислить. Сколько времени потребуется сигналу на преодоление этого расстояния. Скорость света равна примерно 186000 миль в секунду, следовательно, для вычисления времени задержки сигнала (времени между отправкой и приемом сигнала) необходимо разделить расстояние на скорость света. Значение времени должно быть выведено в секундах и минутах.

Пошаговая инструкция

1. Создайте новый файл и назовите его `Mars.cs`.
2. Поскольку значение времени задержки сигнала будет содержать дробные части, для него необходимо использовать значения с плавающей точкой. Перечислим переменные, которые будут использоваться в программе:

```
double distance;
double lightspeed;
```

```
double delay;
double deiay_in_min;
```

3. Задайте переменным `distance` и `lightspeed` начальные значения:

```
distance = 34000000; // 34000000 миль
lightspeed = 186000;  // 186000 миль в секунду
```

4. Для вычисления времени задержки сигнала разделите значение переменной `distance` на значение переменной `lightspeed`. При этом будет получено значение задержки сигнала в секундах. Присвойте это значение переменной `delay` и выведите результаты следующим образом:

```
delay = distance / lightspeed;

Console.WriteLine("Расстояние между Землей и Марсом сигнал " +
                  "пройдет за \n" + delay + " секунд.");
```

5. Для получения значения результата в минутах разделите значение результата в секундах, который содержится в переменной `delay`, на 60; выведите результат на экран, используя следующие строки кода:

```
delay_in_min = delay / 60;

Console.WriteLine("Время задержки сигнала составляет " + delay_in_min
                  + " минут.");
```

Ниже приведен полный листинг программы `Mars.cs`:

```
/*
  Проект 2-1

  "Разговор с Марсом"

  Назовите этот файл Mars.cs
*/

using System;

 class Mars {
  public static void Main ()   {
     double distance;
     double lightspeed;
     double delay;
     double delay_in_min;

     distance = 34000000; // 34000000 миль
     lightspeed = 186000; // 186,000 миль в секунду

     delay = distance / lightspeed;

     Console.WnteLine("Расстояние между Землей и Марсом сигнал" +
                 "пройдет за \n" + delay + " секунд.");

     delay_in_min = delay / 60;

     Console.WnteLine("Время задержки сигнала составляет " +
             delay_in_min + " минут.");
```

6. Скомпилируйте и запустите программу. Результат ее выполнения будет следующим:

```
Расстояние между Землей и Марсом сигнал пройдет за 182.795698924731 секунд.
Время задержки сигнала составляет 3.04659498207885 минут.
```

7. Конечно, большинство людей в своей работе не сталкиваются с числами, имеющими слишком много десятичных разрядов. Для улучшения читабельности выводимого результата программы замените соответствующие операторы `WriteLine();` представленными ниже:

```
Console.WriteLine("Расстояние между Землей и Марсом сигнал " +
                      "пройдет за \n{0:###.###} секунд.", delay);

Console.WriteLine("Время задержки сигнала составляет {0:###.###} минут.",
delay_in_min);
```

8. Повторно скомпилируйте и запустите программу. Теперь на экран будет выведен отформатированный результат вычислений:

```
Расстояние между Землей и Марсом сигнал пройдет за
182.796 секунд.
Время задержки сигнала составляет 3.047 минут.
```

В этом случае на экран выводятся только три десятичных разряда.

Литералы

В C# литералами (или константами) называют фиксированные значения, представленные в удобной для чтения форме (например, число 100 является литералом). В основном литералы и способы их использования настолько понятны, что мы без особых объяснений применяли их в той или иной форме во всех предыдущих программах. Теперь же расскажем о них более подробно.

В C# литералы могут быть значениями любого типа, от которого зависит способ представления каждого литерала. Как уже говорилось, символьные константы заключаются в одинарные кавычки; так, `'a'` и `'%'` являются символьными константами. Целочисленные литералы специфицируются как числа без дробной части. Например, 10 и -100 являются целочисленными константами. Константы с плавающей точкой требуют при написании использования десятичной точки, за которой следует дробная часть числа. Например, 11.123 является константой с плавающей точкой. В C# разрешается также использовать экспоненциальное представление чисел с плавающей точкой. Поскольку C# - язык со строгим контролем типов, литералы также определяются как

принадлежащие к какому-нибудь типу. Чтобы не возникало затруднений при определении типа каждого числового литерала, в C# существуют специальные правила.
Типом целочисленных литералов является наименьший целочисленный тип, начиная с `int`, который способен хранить данное число. Следовательно, в зависимости от величины числа типом
целочисленных литералов может быть `int`, `uint`, `long` или `ulong`. Литералы с плавающей точкой имеют тип `double`.

Если вас не устраивает тип, определенный в C# по умолчанию, вы можете задать нужный тип литерала явно посредством добавления суффикса. Чтобы указать тип

литерала long, добавьте к числу букву l или L (например, число 20 имеет тип int, а 12L - тип long). Для спецификации беззнакового целочисленного литерала добавьте букву u или U (например, число 100 имеет тип int, а 100U - тип uint). Для спецификации длинного беззнакового целого прибавьте к константе суффикс ul или UL (например, литерал 984375UL имеет тип ulong). Чтобы специфицировать литерал типа float, добавьте к константе символ f или F (например, литерал 10.19F имеет тип float). Для спецификации литералов типа decimal добавьте к значению букву m или M (9.95M является литералом, имеющим тип decimal). Хотя целочисленные литералы по умолчанию создаются как значения, имеющие тип int, uint,

long или ulong, они могут быть присвоены переменным типа byte, sbyte, short или ushort, но только в том случае, если присваиваемое значение вообще может представляться этим типом. Целочисленный литерал всегда может быть присвоен переменной типа long.

Шестнадцатеричные литералы

Иногда в программировании вместо десятичной легче применять шестнадцатеричную систему, использующую цифры от 0 до 9 и символы от A до F, которым присвоены значения от 10 до 15 соответственно. Например, шестнадцатеричное число 10 соответствует 16 в десятичной системе исчисления. Учитывая достаточно частое применение шестнадцатеричных чисел, C# позволяет использование целочисленных констант в шестнадцатеричном формате. Шестнадцатеричные литералы должны начинаться с символов 0x (ноль и x). Приведем несколько примеров:

```
count = 0xFF; // 255 в десятичной системе
incr = 0x1a; // 26 в десятичной системе
```

Символьные escape-последовательности

В C# большинство выводимых символьных констант можно заключать в одинарные кавычки, но применение некоторых символов (например, символа возврата каретки) является проблематичным Кроме того, часть символов (например, символы одинарных или двойных кавычек) имеют в C# специальное значение, поэтому их прямое использование недопустимо. По этим причинам в C# предусмотрены специальные *escape-последовательности,* которые перечислены в табл. 2.2 Эти последовательности используются вместо символов, которые они представляют.

Например, таким образом переменной присваивается символ табуляции:

```
ch = '\t';
```

А так переменной ch присваиваются одинарные кавычки:

```
ch = '\'';
```

Таблица 2.1. Символьные escape-последовательности

Escape-последовательности	Описание
\a	Предупреждение (звонок)
\b	Возврат на одну позицию (backspace)
\f	Переход на новую страницу (formfeed)
\n	Переход на новую строку (linefeed)
\r	Возврат каретки

Escape-последовательности	Описание
\t	Горизонтальная табуляции
\v	Вертикальная табуляция
\0	Ноль
\'	одинарные кавычки
\"	Двойные кавычки
\\	Символ обратной косой черты

Строковые литералы

В C# поддерживается еще один тип литералов - строка, являющаяся набором символов, заключенных в двойные кавычки. Ниже приведен пример строки:

```
"Небольшой проверочный текст."
```

Вы уже встречали примеры таких строк, применявшиеся во многих операторах `WriteLine();` предыдущих программ.

Кроме обычных символов строковые литералы могут содержать одну или несколько escape-последовательностей. В качестве примера рассмотрим программу, которая использует escape-последовательности `\n` и `\t`.

```
// В программе демонстрируется использование escape-последовательностей
// в строковых литералах

using System,

class StrDemo {
  public static void Main() {
    Console.WriteLine("Первая строка.\nВторая строка.");
    Console.WriteLine("A\tB\tC");
    Console.WriteLine("D\tE\tF");
  }
}
```

Использование escape-последовательности \n для перехода на новую строку.

Использование табуляции для выравнивания выводимых символов.

Эта программа выводит следующие строки

```
Первая строка.
Вторая строка.
A       B       C
D       E       F
```

Ответы профессионала

Вопрос. Я знаю, что в языке C++ разрешается специфицировать литералы в восьмеричной системе. Разрешены ли восьмеричные литералы в C#?

Ответ. Нет, в C# литералы можно специфицировать только в десятичной и шестнадцатеричной форме. В современном программировании восьмеричная форма встречается редко.

Заметьте, что escape-последовательность `\n` используется для перехода на новую строку. Чтобы разбить выводимые данные на несколько строк, не обязательно использовать

множество операторов `WriteLine();`, достаточно поместить escape-последовательность `\n` в пределах длинной строки там, где должна начинаться новая строка.

В дополнение к вышеизложенному добавим, что в C# имеется еще возможность специфицирования копирующегося строкового литерала. Он начинается с символа `@`, за которым следует строка в кавычках. Содержимое строкового литерала, взятого в кавычки, принимается без модификации и может охватывать две и более строки. Следовательно, в этом случае можно использовать символы перехода на новую строку, табуляции и так далее без применения escape-последовательностей. Единственное исключение состоит в том, что при необходимости получения в выводимых данных символа двойных кавычек (") их нужно ввести дважды подряд (""). Следующая программа демонстрирует использование копирующихся строковых литералов.

```
// В программе демонстрируется использование копирующихся строковых литералов.

using System;

class Verbatim {
  public static void Main() {
    Console.WriteLine(@"Это копирующийся строковый литерал,
который охватывает
несколько строк.
");
    Console.WriteLine(@"Здесь также используется табуляция:
1   2   3   4
5   6   7   8
");
    Console.WnteLine(@"Программисты говорят; ""C# — интересный язык.""");

  }
}
```

Этот копирующийся строковый литерал содержит вложенные символы перехода на новую строку.

В этом копирующемся строковом литерале содержатся также символы табуляции.

Результат выполнения этой программы выглядит следующим образом:

```
Это копирующийся строковый литерал,
который охватывает
несколько строк.

Здесь также используется табуляция:
1      2      3     4
5      6      7     8

Программисты говорят: "C# - интересный язык."
```

Обратите внимание на то, что копирующиеся строковые литералы выводятся в программе точно так же, как они были введены. Но если копирующийся литерал занимает несколько строк и символы выводятся на экран с самого начала строки, то нарушается читабельность программы, поскольку не соблюдается стиль ввода кода с использованием отступов. Поэтому мы не используем копирующиеся строковые литералы в программах данной книги, но вы должны знать, что это прекрасное решение во многих случаях, когда требуется форматирование.

Минутный практикум

1. Каков тип литерала 10? Каков тип литерала 10.0?
2. Специфицируйте значение 100 как `long`. Специфицируйте значение 100 как `uint`.
3. Что такое @"testing"?

Ответы профессионала

Вопрос. Является ли символьным литералом строка, содержащая одиночный символ (например, `"k"` и `'k'`)?

Ответ. Нет. Не путайте строки с символами. Символьный литерал представляет единичный символ типа char. Строка, содержащая только одну букву, все равно остается строкой. Хотя строки состоят из символов, они имеют другой тип.

Переменные и их инициализация

О переменных мы уже коротко говорили в главе 1. Вам известно, что переменные объявляются с помощью оператора, имеющего следующий синтаксис:

```
type var-name;
```

где `type` - это тип переменной, а `var-name` - ее имя. Вы можете объявлять переменные любого действительного типа, включая рассмотренные выше простые типы. При создании переменной создается экземпляр ее типа, следовательно, возможности переменной определяются ее типом. Например, переменная типа `bool` не может использоваться для хранения значений с плавающей точкой. Более того, при объявлении переменной ее тип фиксируется и изменять его нельзя.
Например, переменная типа `int` не может превратиться в переменную типа `char`.

Все переменные в C# должны быть объявлены до их использования. Это необходимо, поскольку компилятору нужно знать тип данных, которые содержит переменная, до того как он начнет компилировать операторы, использующие эту переменную.

В C# определено несколько различных типов переменных. Переменные, которые использовались в наших программах до этого момента, называются локальными переменными, поскольку они объявлялись в области видимости метода.

Инициализация переменной

Прежде чем использовать переменную, вы должны присвоить ей значение. Сделать это можно двумя способами. Можно присвоить значение переменной в отдельном операторе после ее объявления. А можно сделать это в одном операторе, затем объявить тип переменной и присвоить ей значение, для чего ввести после имени переменной символ знака равенства (оператор присваивания), а за ним - присваиваемое значение. Синтаксис оператора инициализации показан ниже.

```
type var = value;
```

Здесь value — это значение, которое присваивается переменной во время ее создания. Значение должно быть совместимо с указанным типом.

Приведем несколько примеров объявления переменных с одновременным присваиванием им значений

```
int count = 10; // переменной count задается начальное значение 10
char ch = 'X' // переменной ch присваивается символ X
float f = 1.2F; // переменной f присваивается значение 1.2
```

При объявлении двух и более переменных одною типа можно разделить переменные в списке запятыми и присвоить одной или нескольким из них начальное значение. Например:

```
int a, b - 8, c - 19, a; // переменным b и c присвоено начальное значение
```

В этом случае инициализированы только переменные b и c

Динамическая инициализация

В предыдущих примерах переменным присваивались только конкретные числовые значения (то есть значения, которые не были получены в результате вычисления какого-либо выражения) Но в C# можно инициализировать переменные и динамически, используя любое действительное выражение. Например, ниже приведена короткая программа, которая вычисляет объем цилиндра по заданному радиусу основания и высоте:

```
// В программе демонстрируется использование
// динамической инициализации переменной.

using Syster;

class DynInit {
    public static void Main() {
      double radius = 4, height = 5;

      // Динамически инициализируется переменная volume
      double volume = 3.1416 * radius * radius * height;

      Console.WriteLine("Объем цилиндра равен: " + volume);
  }
}
```

Переменная volume инициализируется динамически во время выполнения программы.

В этой программе объявляются три локальные переменные - radius, height и volume. Первые две, radius и height, инициализированы конкретными значениями. Переменная volume инициализируется динамически - ей присваивается значение, соответствующее объему цилиндра. Ключевым моментом является то, что в выражении инициализации может быть использован любой элемент, который уже был до этого объявлен (то есть действительный на время инициализации данной переменной). Таким элементом может быть метод, другая переменная или литерал.

Область видимости и время жизни переменных

Все переменные, которые мы использовали до этого момента, объявлялись в начале метода Main(). Но в C# можно объявлять локальные переменные и внутри блока.

Как уже говорилось в главе 1, блок начинается открывающейся фигурной скобкой и заканчивается закрывающей фигурной скобкой. Он определяет *область видимости*, которая зависит от того, имеет ли данный блок вложенные блоки. Каждый раз, создавая блок, вы создаете новую область видимости, определяющую время жизни объявляемых переменных.

Наиболее важными областями видимости в C# являются те, которые определяются классом и методом. На данном этапе мы рассмотрим только область видимости, определяемую методом.

Область видимости, определяемая методом, начинается с открывающей фигурной скобки. Однако если этот метод имеет параметры, они также включаются в область видимости метода. Общее правило состоит в том, что объявленная внутри области видимости переменная является невидимой (то есть недоступной) для кода, который определен за этой областью. Следовательно, при объявлении переменной в пределах ее области видимости вы локализуете эту переменную и защищаете ее от неразрешенного доступа и модификации. Можно сказать, что правила, касающиеся области видимости, обеспечивают фундамент для инкапсуляции.

Область видимости может быть вложенной. Например, каждый раз, создавая блок кода, вы создаете новую вложенную область видимости, и при этом внешняя область становится видимой для вложенной области. Это означает, что объекты, объявленные во внешней области видимости, будут видимы для кода из внутренней области видимости, но объекты, объявленные во внутренней области видимости, не будут видимы для кода внешней области видимости.

Чтобы понять это правило, рассмотрим следующую программу:

```
// В программе демонстрируется зависимость возможности доступа к
// переменной от области видимости, в которой она была объявлена.

using System;

class ScopeDemo {
  public static void Main() {
    int x; // Переменная x известна всему коду
           // в пределах метода Main().

    x = 10;
    if(x == 10) { //Создается новая область видимости.
      int y = 20; // Эта переменная будет видна только в рамках
                  // этого блока.

      // В данном блоке видны обе переменные, x и y.
      Console.WriteLine("Видны x и y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Ошибка! Переменная y здесь не видна!

    // Выполняется доступ к переменной x, которая была объявлена
    // в этой области видимости.
    Console.WriteLine("Значение переменной x равно: " + x);
  }
}
```

y - это переменная, объявленная во вложенной области видимости, поэтому здесь она не видна.

Как указано в комментариях, объявленная в методе `Main()` переменная `x` доступна всему коду в пределах этого метода. Переменная `y` объявляется в рамках блока `if`. Поскольку блок определяет область видимости, переменная y является видимой только в пределах самого блока. Вот почему использование переменной `y` в строке `y = 100;` вне своего блока является ошибкой, о чем и говорится в комментарии. Если вы удалите начальный символ комментария, произойдет ошибка компилирования, так как переменная y невидима за пределами своего блока. В рамках блока `if` может использоваться переменная `x`, поскольку в блоке (вложенной области видимости) код имеет доступ к переменным, объявленным во внешней области видимости.

Внутри блока переменная может быть объявлена в любом месте, после чего она становится действительной. Следовательно, если вы определите переменную в начале метода, она будет доступна всему коду в рамках метода. И наоборот, если вы объявите переменную в конце блока, то не сможете использовать ее до момента объявления, поскольку код не будет иметь к ней доступа.

Обратите внимание на еще один важный момент: переменные создаются при их объявлении в какой-либо области видимости (при входе в данную область) и уничтожаются при выходе из этой области. Это означает, что переменная не будет сохранять свое значение при выходе из своей области видимости, то есть переменная, объявленная в пределах метода, не будет хранить свои значения между вызовами этого метода. Также переменная, объявленная в рамках блока, будет терять свое значение при выходе из блока. Следовательно, время жизни переменной ограничено ее областью видимости.

Если при объявлении переменной одновременно происходит ее инициализация, то переменная будет повторно инициализироваться каждый раз при входе в блок, в котором она объявлена. В качестве примера рассмотрим следующую программу:

```
// В программе демонстрируется использование правила,
// определяющего время жизни переменной.

using System;

class VarInitDemo {
  public static void Main() {
    int x;

    £or(x - 0; x < 3; x++)   {
      int y = —1;  // Переменная y инициализируется каждый раз
                   // при входе в блок.
      Console.WriteLine("Значение переменной y равно: " + y);
                        // Всегда выводится значение -1.
      y = 100;
      Console.WriteLine("Теперь значение переменной y равно: " + y);
    }
  }
```

Ниже показан результат выполнения этой программы:

```
Значение переменной y равно: -1
Теперь значение переменной y равно: 100
Значение переменной y равно: -1
Теперь значение переменной y равно: 100
Значение переменной y равно: -1
Теперь значение переменной y равно:100
```

Как видите, переменная `y` инициализируется значением -1 каждый раз при входе во внутренний цикл `for`, затем ей присваивается значение 100, которое утрачивается при завершении цикла.

Заметим, что между языками C# и C есть отличие в работе с областями видимости

Несмотря на то, что блоки могут быть вложенными, переменная, объявленная во внутренней области видимости, не может иметь такое же имя, как переменная, объявленная по внешней области видимости Следующая программа, и которой делается попытка объявить две переменные с одинаковыми именами, не будет скомпилирована:

```
/*
   В этой программе делается попытка объявить во внутренней области
   видимости переменную с тем же именем, что и у переменной,
   объявленной во внешней области видимости.

   *** эта программа не будет скомпилирована. ***
*/

using System;

class NestVar {
  public static void Main() {
    int count;

    for(count = 0; count < 10; count = count+1) {
      Console.WriteLine("Счет проходов цикла: " + count);

      int count; // Это объявление недействительно'!'
      for (count = 0; count < 2; count++)
        Console.WnteLine("В программе есть ошибка!");
    }
  }
}
```

Переменную `count` объявлять нельзя, поскольку переменная с таким именем уже объявлена и методе `Main()`

Если у вас есть опыт программирования на C/C++, то вам известно, что в этих языках нет ограничений для имен переменных, объявляемых во внутренней области видимости. То есть в C/C++ объявление еще одной переменной `count` в цикле `for` является допустимым, но при этом данное объявление скроет внешнюю переменную. Чтобы скрытие переменной не приводило к ошибкам программирования, разработчики C# запретили такое действие.

Минутный практикум

1. Что определяет область видимости?
2. Где в блоке могут объявляться переменные?
3. Когда в блоке создается переменная? Когда она уничтожается?

1. Область видимости определяет возможность доступа к переменной и время ее жизни. Область видимости создается при создании блока.
2. Переменные могут быть объявлены в любом месте блока
3. Внутри блока переменная создается при объявлении этой переменной. Переменная уничтожается при выходе из блока.

Операторы

В C# предусмотрен большой выбор операторов. Оператором является символ, сообщающий компилятору о необходимости выполнения специфической математической или логической операции. В C# существуют четыре основных класса операторов (арифметические, побитовые, логические и операторы сравнения), а также несколько дополнительных операторов для обработки некоторых специальных ситуаций. В этой главе будут рассмотрены арифметические, логические операторы и операторы сравнения. Кроме того, мы расскажем об операторе присваивания. Побитовые и другие специальные операторы будут рассмотрены позже.

Арифметические операторы

В C# определены следующие арифметические операторы.

Оператор	Значение
+	Сложение
-	Вычитание (также унарный минус)
*	Умножение
/	Деление
%	Взятие по модулю (остаток от деления)
++	Инкремент
--	Декремент

Арифметические операторы сложения (+), вычитания (-), умножения (*) и деления (/) работают в C# так же, как в любом другом языке программирования, они могут применяться к любым встроенным числовым типам данных.

Хотя действие арифметических операторов вам хорошо известно, некоторые специальные ситуации требуют пояснения. При применении оператора деления к целочисленным данным любой остаток будет отброшен (например, в целочисленном делении 10/3 будет равно 3). Остаток этого деления можно получить при использовании оператора взятия по модулю (%). Он работает в C# так же, как в других языках, - получает остаток от целочисленного деления (так, 10 % 3 будет равно 1). В C# оператор взятия по модулю может применяться как к целочисленным данным, так и к значениям с плавающей точкой (10.0 % 3.0 будет также равно 1), в отличие от C/C++, где операцию взятия по модулю разрешено производить только с целочисленными типами данных.

Следующая программа демонстрирует применение оператора взятия по модулю:

```
  // В программе демонстрируется использование оператора взятия по модулю.

using System;

class ModDemo {
 public static void Main() {
        int iresult, irem;
         double dresult, drem;

   iresult = 10 / 3;
   irem = 10 % 3;
```

```
     dresult = 10.0 / 3.0;
      drem = 10.0 % 3.0;

    Console.WriteLine("Результат целочисленного деления 10/3 равен: "
        +iresult + "\nостаток равен: " + irem);
    Console.WriteLine("Результат деления чисел, имеющих тип double, - "+
         " 10.0/3.0 равен: \n" + dresult + "\n остаток равен: " + drem);
  }
}
```

Программа выводит следующий результат:

```
  Результат целочисленного деления 10/3 равен: 3
   остаток равен: 1
Результат деления чисел, имеющих тип double,  - 10.0/3.0 равен:
3.33333333333333 остаток равен: 1
```

Как видите, оператор взятия по модулю возвращает значение 1 как при работе с целочисленными операндами, так и при работе с данными, имеющими тип `double`.

Операторы инкремента и декремента

Об операторах инкремента (`++`) и декремента (`--`) мы уже упоминали в главе 1. У этих операторов есть некоторые специальные свойства, которые делают их очень полезными при задании алгоритма изменения переменной цикла (например, в цикле `for`). Рассмотрим, как функционируют операторы инкремента и декремента.

Оператор инкремента добавляет единицу к своему операнду, а оператор декремента вычитает единицу. Следовательно, оператор

```
x = x + 1;
```

выполняет ту же самую последовательность действий, что и оператор

```
X++;
```

а оператор

```
x = x - 1;
```

выполняет ту же самую последовательность действий, что и оператор

```
X--;
```

Оба ли оператора могут указываться либо до операнда (как префикс), либо после (как постфикс). Например, оператор

```
x = x + 1
```

можно записать так:

```
        ++x; // оператор указывается до операнда
```

или так:

```
x++; // оператор указывается после операнда
```

В этом примере оператор инкремента указан двумя способами, между которыми нет никакой смысловой разницы. Но когда оператор инкремента или декремента используется как часть большего выражения, то от расположения оператора по отношению к операнду зависит алгоритм выполнения блока кода. Если оператор указывается перед операндом, C# увеличивает значение переменной (операнда) до того, как эта переменная будет использована оставшейся частью выражения. Если оператор указывается после операнда, то C# увеличивает значение переменной после того, как

оставшаяся часть выражения использует эту переменную в своих операциях. Рассмотрим следующий пример:

```
x = 10;
    y = ++x;
```

В этом случае переменной y будет присвоено значение 11.

Но если код будет написан так:

то вначале переменной `y` будет присвоено значение 10, а затем значение переменной `x` будет увеличено на единицу. В обоих случаях значение переменной `x` будет увеличено до 11, различие состоит только во времени, когда это произойдет.

Благодаря названным свойствам операторы инкремента и декремента широко используются при формировании алгоритмов.

Операторы сравнения и логические операторы

В терминах оператор сравнения и логический оператор слово сравнение означает оценку одного значения по сравнению с другим, а слово логический - способ, которым можно связать значения `t &e` и `false` (истина и ложь) в выражении. Поскольку результатом выполнения операторов сравнения являются булевы значения, эти операторы часто работают совместно с логическими операторами. Поэтому их можно рассматривать вместе.

Ниже представлены операторы сравнения.

Оператор	Значение
==	Равно
!=	Не равно
>	Больше чем
<	Меньше чем
>=	Больше или равно
<=	Меньше или равно

Далее перечислены логические операторы.

Оператор	Значение
&	AND (И)
|	OR (ИЛИ)
^	XOR (исключающее ИЛИ)
&&	Short-circuit AND (быстрый оператор И)
||	Short-circuit OR (быстрый оператор ИЛИ)
!	NOT (НЕ)

Результатом выполнения операторов сравнения и логических операторов являются значения типа `bool`.

В C# операторы == и != могут применяться ко всем объектам для их сравнения на предмет равенства или неравенства. Операторы сравнения <, >, <=, >= применимы только к перечисляемым типам данных, которые упорядочены в своей структуре (например, упорядоченная структура чисел 1, 2, 3 и так далее или упорядоченные

символы букв алфавита). Следовательно, все операторы сравнения могут применяться ко всем числовым типам данных. Однако значения типа `bool` могут сравниваться только на предмет равенства или неравенства, поскольку значения `true` и `false` не упорядочены. Например, выражение `true > false` в языке C# не имеет смысла.

Для логических операторов операнды должны иметь тип `bool`. Результатом логических операций также являются значения, имеющие тип `bool`. Логические операторы `&`, `|`, `^` и `!` поддерживают базовые логические операции AND, OR, XOR и NOT соответственно, результаты выполнения которых соответствуют значениям, приведенным в следующей таблице истинности.

p	**q**	**p&q**	**p**	**q**	**p^q**
false	false	false	false	false	true
true	false	false	true	true	false
false	true	false	true	true	true
true	true	true	true	false	false

Например, из таблицы видно, что результатом выполнения оператора `XOR` будет значение `true`, если только один его операнд имеет значение `true`.

Рассмотрим программу, в которой демонстрируется несколько операторов сравнения и логических операторов:

```
// В программе демонстрируется использование логических операторов
// и операторов сравнения.

using System;

class RelLogOps {
  public static void Main() {
    int i, j;
    bool b1, b2;

    i = 10;
    j = 11;
         if(i < j) Console.WriteLine("i < j");
    if(i <= j) Console.WriteLine("i <= j");
    if(i != j) Console.WriteLine("i != j");
    if(i == j) Console.WriteLine("Эта строка не будет выведена.");
         if(i >= j) Console.WriteLine("Эта строка не будет выведена.");
         if(i > j) Console.WriteLine("Эта строка не будет выведена.");

    b1 = true;
       b2 = false;
    if(b1 & b2) Console.WriteLine("Эта строка не будет выведена.");
    if(!(b1 & b2)) Console.WriteLine("Результатом вычисления"+
                  " выражения !(b1 & b2) будет значение true.");
    if(b1 | b2) Console.WriteLine("Результатом вычисления"+
                  " выражения b1 | b2 будет значение true.");
    if(b1 ^ b2) Console.WriteLine("Результатом вычисления"+
                  " выражения b1 ^ b2 будет значение true.");
   }
}
```

Результат выполнения программы следующий:

```
i < j
i <= j
i != j
Результатом вычисления выражения !(b1 & b2) будет значение true.
Результатом вычисления выражения b1 | b2 будет значение true.
Результатом вычисления выражения b1 ^ b2 будет значение true.
```

Быстрые логические операторы

В C# предусмотрены специальные *быстрые* версии логических операторов `AND` и `OR`, при использовании которых программа будет выполняться быстрее. Рассмотрим, как они работают. Если при выполнении оператора `AND` первый операнд имеет значение `false`, результатом будет `false`, независимо от значения второго операнда. Если при выполнении оператора `OR` первый операнд имеет значение `true`, результатом будет значение `true`, независимо от значения второго операнда. Следовательно, в этих случаях нет необходимости оценивать второй операнд, и соответственно программа будет выполняться быстрее.

Быстрый оператор `AND` обозначается символом `&&`, а быстрый оператор `OR` - символом `||`. Единственная разница между обычной и быстрой версиями операторов состоит в том, что обычные операторы всегда оценивают и первый, и второй операнды, а быстрые операторы оценивают второй операнд только при необходимости.

Ниже приведена программа, в которой демонстрируется использование быстрого оператора `AND`. Программа определяет, является ли значение переменной `d` кратным значению переменной `n`. Это осуществляется с помощью оператора взятия по модулю. Если остаток деления `n/d` равен нолю, то значение переменной `d` является делителем значения переменной `n`. Однако поскольку операция взятия по модулю предусматривает выполнение деления, в программе используется быстрый оператор `AND` для предотвращения деления на ноль.

```
// В программе демонстрируется использование быстрого оператора AND.

using System;

class SCops {
  public static void Main() {
    int n, d;

     n = 10;
    d = 2;
    if(d != 0 && (n % d) == 0)
      Console.WriteLine(d + " является делителем числа " + n);

    d = 0; // Теперь присваиваем переменной d значение 0.

    // Поскольку переменная d имеет значение 0, второй операнд
    // не оценивается.
    if(d != 0 && (n % d) == 0)
      Console.WriteLine(d + " является делителем числа " + n);

    /* Теперь в выражении условия оператора if будет использован
       обычный оператор AND, что приведет к делению на ноль.
    */
```

Применение быстрого оператора предотвращает деление на ноль.

```
    if(d != 0 & (n % d) == 0) ← Оцениваются оба операнда, что приводит к делению на ноль.
      Console.WriteLine(d + " является делителем числа " + n);
  }
}
```

Для предотвращения деления на ноль вначале оператор `if` проверяет, не равно ли нолю значение переменной `d`. Если равно, то быстрый оператор `AND` останавливает дальнейшее выполнение оператора и деление по модулю не выполняется. Поскольку в программе в первом случае значение переменной `d` равно 2, выполняется операция взятия по модулю. Во втором случае переменной `d` присваивается заведомо неприемлемое значение 0 и операция взятия по модулю не выполняется, в результате чего предотвращается деление на ноль. Наконец, в третьем случае используется обычный оператор `AND`. При этом оцениваются оба операнда, что приводит к ошибке выполнения программы при попытке деления на ноль.

Проект 2-2. Вывод таблицы истинности логических операторов

`LogicOpTable.cs`

В этом проекте рассматривается создание программы, которая выводит на экран таблицу истинности логических операторов. В программе были использованы элементы C# - одна из escape-последовательностей и логические операторы, описанные в этой главе. В проекте также демонстрируется различие в приоритетности между арифметическим оператором + и логическими операторами.

Пошаговая инструкция

1. Создайте новый файл и назовите его `LogicalOpTable.cs`.
2. Для выравнивания столбцов таблицы используйте escape-последовательность \t, которая вставляет символы табуляции в каждую строку вывода. Например, представленный ниже оператор `WriteLine()` выводит заголовок для таблицы:
   ```
   Console.WriteLine("P\tQ\tAND\tOR\tXOR\tNOT");
   ```
3. Для каждой следующей строки таблицы используйте символы табуляции, чтобы поместить результат выполнения каждой операции под соответствующим заголовком.
4. Ниже представлен полный код программы, который нужно ввести в файл `LogicalOpTable.cs`.

```
/*
   Проект 2-2

   Выводит на печать таблицу истинности логических операторов.
*/

using System;

class LogicalOpTable {
  public static void Main() {

    bool p, q;
```

```
Console.WriteLine("P\tQ\tAND\tOR\tXOR\tNOT");
p = true; q = true;
Console.Write(p + "\t" + q +"\t");
Console.Write((p&q) + "\t" + (p|q) + "\t");
Console.WriteLine((p^q) + "\t" + (!p));
p = true; q = false;
Console.Write(p + "\t" + q +"\t");
Console.Write((p&q) + "\t" + (p|q) + "\t");
Console.WriteLine((p^q) + "\t" + (!p));
p = false; q = true;
Console.Write(p + "\t" + q +"\t");
Console.Write((p&q) + "\t" + (p|q) + "\t");
Console.WriteLine((p^q) + "\t" + (!p));
p = false; q = false;
Console.Write(p + "\t" + q +"\t");
Console.Write((p&q) + "\t" + (p|q) + "\t");
Console.WriteLine((p^q) + "\t" + (!p));
  }
}
```

5. Скомпилируйте и запустите программу. В результате ее выполнения будет выведена следующая таблица:

```
P       Q       AND     OR      XOR     NOT
true    true    true    true    false   false
true    false   false   true    true    false
false   true    false   true    true    true
false   false   false   false   false   true
```

6. Обратите внимание, что логические операторы, указываемые в методах `Write()` и `WriteLine()` в качестве параметров, заключены в круглые скобки. Это необходимо, поскольку в C# оператор + имеет более высокий приоритет, чем логические операторы.

7. Попытайтесь самостоятельно модифицировать программу таким образом, чтобы на экран выводились символы `1` и `0` вместо слов `true` и `false`. Возможно, это задача окажется достаточно сложной!

Ответы профессионала

Вопрос. Зачем в C# нужны обычные операторы `AND` и `OR`, если в некоторых случаях быстрые операторы являются более эффективными?

Ответ. Иногда в операторах `AND` и `OR` требуется оценить оба операнда. Это иллюстрируется следующей программой.

```
// Еще одна программа, в которой демонстрируется разница в
 // применении быстрого и обычного операторов AND.

    using System;

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

```
int i;
i = 0;
/* В выражении условия оператора if к переменной i будет
применена операция инкремента, даже если условие окажется ложным.
*/
if(false & (++i < 100))
Console.WriteLine("Эта строка не будет выведена.");
// Следующим оператором будет выведено
// значение переменной i, равное 1.
Console.WriteLine("Если оператор инкремента будет выполнен,"+
" переменная i примет значение: " + i);
/* В этом случае к переменной i не будет применена операция
инкремента, поскольку используется быстрый оператор AND.
*/
if(false && (++i < 100))
Console.WriteLine("Эта строка не будет выведена.");
// Следующим оператором вновь будет выведено
// значение переменной i, равное 1.
Console.WriteLine("Если оператор инкремента будет выполнен,"+
" переменная i примет значение: " + i);
}
}
```

Комментарии указывают, что в выражении условия первого оператора if к переменной i будет применена операция инкремента независимо от истинности выражения. Но при использовании быстрого оператора значение переменной i не будет увеличено на единицу, поскольку первый операнд имеет значение false. То есть этот пример показывает, что если в коде необходимо выполнить проверку операнда, находящегося справа от оператора AND, вы должны использовать обычную форму этого оператора (то же касается оператора OR).

Минутный практикум

1. Какие действия производит оператор взятия по модулю (%)? К каким типам данных он может быть применен?
2. Какие типы данных могут использоваться в качестве операндов логических операторов?
3. Всегда ли быстрый оператор оценивает оба своих операнда?

Оператор присваивания

Мы уже неоднократно использовали оператор присваивания в примерах программ этой книги, а теперь рассмотрим его более подробно. Оператор присваивания указывается как одинарный знак равенства (=). В C# он работает практически так же, как

---

1. Оператор взятия по модулю возвращает остаток целочисленного деления. Он может быть применен ко всем числовым типам данных
2. Операнды логических операторов должны иметь тип `bool`
3. Нет, быстрый оператор оценивает второй операнд лишь нельзя определить, исходя из истинности

в любом другом языке программирования. Оператор присваивания имеет следующий синтаксис:

```
var = expression;
```

Тип *переменной* должен быть совместим с типом *выражения*

Оператор присваивания имеет одно очень интересное свойство, с которым вы, возможно, еще не знакомы, — он позволяет создавать «цепочку присваивании». В качестве примера рассмотрим фрагмент кода:

```
int x, y, z;
x = y = z = 100; // Переменным x, y, z присваивается значение 100.
```

В приведенном выше фрагменте с помощью одного оператора переменным `x`, `y` и `z` присваивается значение 100. Такая последовательность переменных и операторов допускается, поскольку оператор = присваивает переменной, находящейся слева от него, значение выражения, находящегося справа. Следовательно, выражение `z = 100` будет иметь значение 100, которое присваивается переменной `y`, а затем — переменной `x`. Используя «цепочку присваивании», можно одним значением легко инициализировать группу переменных.

Составные операторы присваивания

В C#, как и в C/C++, имеются составные операторы присваивания, в которых арифметические операторы совмещены е операторами присваивания. Рассмотрим применение составного оператора присваивания на примере. Выражение

```
x = x + 10;
```

может быть записано с использованием составного оператора присваивания:

```
x += 10;
```

Пара операторов += указывает, что компилятор должен присвоить неременной `x` значение выражения `x + 10`.

Приведем еще один пример. Выражение

```
x = x - 100;
```

можно записать как

```
x -= 100;
```

Оба оператора присваивают переменной `x` значение `x - 100`

Для всех логических операторов (то есть операторов, требующих два операнда) существуют составные операторы присваивания. Синтаксис этих операторов следующий:

```
var op = expression;
```

Таким образом, в C# имеются следующие составные операторы присваивания:

```
+=      -=      *=      /=
%=      &=      |=      ^=
```

Составные операторы присваивания в сравнении с их «обычными» аналогами имеют два преимущества. Во-первых, они более компактны, а во-вторых, их использование позволяет ускорить компилирование кода (так как операнд оценивается только один раз).

По этим причинам составные операторы присваивания часто используются в профессионально написанных C#-программах.

Преобразование типа в операциях присваивания

В программировании часто применяется присваивание значения переменной одного типа переменной другого типа. Например, вы можете присвоить значение переменной типа `int` переменной типа `float`, как показано ниже:

```
int i;
float f;

i = 10;
f = i; // присвоение значения типа int переменной типа float
```

Если в операции присваивании используются совместимые типы данных, то тип переменной, находящейся справа от оператора, автоматически преобразуется в тип переменной, находящейся слева от него. Следовательно, в предыдущем примере тип переменной `i` преобразуется в тип `float`, а затем значение переменной `i` присваивается переменной `f`. Но поскольку C# является языком со строгим контролем типов, то не все его типы данных являются совместимыми, и следовательно, не все типы преобразований разрешены. Например, типы данных `bool` и `int` несовместимы.

При присваивании одного типа данных переменной другого типа автоматическое преобразование типа происходит, если:

- два типа данных совместимы;
- конечный тип (слева) больше исходного типа (справа).

При выполнении двух этих условий происходит расширяющее преобразование. Например, количества битов, выделенных для типа данных `int`, всегда достаточно для хранения всех действительных значений типа `byte`, а поскольку оба эти типа целочисленные, то к ним может быть применено автоматическое преобразование.

Для выполнения расширяющих преобразований все числовые типы, включая целочисленные данные и данные с плавающей точкой, являются совместимыми. Например, в следующей программе выполняемое преобразование действительно, поскольку преобразование данных типа `long` в `double` является расширяющим и выполняется автоматически.

```
// В программе демонстрируется автоматическое преобразование типа
// значения из long в double.

using System;

class LtoD {
  public static void Main() {
       long L;
    double D;

         L = 100123285L;
    D = L;
          Console.WriteLine("L и D: " + L + " " + D);
  }
```

Автоматическое преобразование типа значения из `long` в `double`.

Хотя существует автоматическое преобразование типа переменной из `long` в `double`, обратное автоматическое преобразование осуществить нельзя, поскольку оно не будет

расширяющим. Следовательно, такая версия предыдущей программы будет недействительной:

```
   // *** Эта программа не будет скомпилирована. ***

    using System;

class LtoD {
  public static void Main() {
     long L;
     double D;

    D = 100123285.0;
    L = D; // Такое преобразование недействительно!!!
    Console.WriteLine("L и D: " + L + " " + D);

  }
}
```

Hl Ьзи автомл ичс<_ки преобразовав переменную imia coublt в inn long

В дополнение к уже указанным ограничениям необходимо добавить, что не существует автоматического преобразования между типами `decimal` и `float` или `double` либо преобразования переменных числового типа в переменные типа `char` или `bool`. Переменные типов `char` и `bool` также несовместимы друг с другом.

Выполнение операции приведения типа между несовместимыми типами данных

Автоматические преобразования очень удобны, но они не отвечают всем потребностям программирования, поскольку в них допускается расширяющее преобразование только между совместимыми типами данных. Во всех других случаях приходится применять операцию приведения типа. *Приведение типа* - это инструкция компилятору для преобразования одного типа данных в другой. Такое преобразование типов данных является явным. Операция приведения типа имеет следующий синтаксис:

```
(target-type) expression
```

Конечный тип (target-type) указывает, к какому типу должно быть приведено выражение. Так, если необходимо преобразовать выражение `x/y` к типу `int`, следует написать:

```
double x, y;
// ...
(int) (x/y)
```

В этом примере приведение типа преобразует результат вычисления выражения к типу `int`, хотя переменные `x` и `y` имеют тип `double`. Круглые скобки, в которые заключено выражение `x/y`, являются обязательными. В противном случае приведение к типу `int` будет применено только для переменной `x`, а не для значения результата деления. Здесь применяется операция приведения типа, поскольку автоматическое преобразование данных типа `double` в тип `int` не может быть выполнено.

Когда в результате приведения типа происходит сужающее преобразование, информация может быть утеряна. Например, если при приведении данных типа `long` в `int` значение переменной, имеющей тип `long`, выйдет за диапазон значений, допустимых для типа `int`, то биты старшего разряда будут удалены, следовательно, информация

будет утеряна. Если значение с плавающей точкой приводится к целочисленному типу, дробные компоненты утрачиваются, поскольку при таком преобразовании они отбрасываются. Например, если значение 1.23 преобразуется в целочисленный тип, результирующим будет значение 1, а дробная часть 0.23 будет утеряна.

Ниже представлена программа, демонстрирующая некоторые виды преобразований с использованием операции приведения типа.

```
// В программе демонстрируется использование операции приведения типа.

using System;

class CastDemo {
  public static void Main() {
double x, y;
byte b;
int i;
char ch;
x = 10.0;
y = 3.0;
i = (int) (x/y); // Выражение, имеющее тип double, приводится
// к типу int.
Console.WriteLine("Целая часть выражения x/y равна: " + i);
i = 100;
b = (byte) i;
Console.WriteLine("Значение переменной b равно: " + b);
i = 257;
b = (byte) i;
Console.WriteLine("Значение переменной b равно: " + b);
b = 88; // Значение ASCII-кода для символа X (англ).
ch = (char) b;
Console.WriteLine("ch: " + ch);
}
}
```

Результат выполнения программы выглядит следующим образом:

```
Целая часть выражения x/y равна: 3
Значение переменной b равно: 100

Значение переменной b равно: 1
    ch: X
```

В этой программе приведение результата вычисления выражения (`x/y`) к типу `int` приводит к отбрасыванию дробной части и потере информации. Затем, когда переменной `b` присваивается значение 100, информация не теряется, поскольку это значение находится в диапазоне допустимых значений, определенных для типа `byte`. Но при попытке присвоить переменной `b` значение 257 происходит потеря информации, потому что число 257 превышает максимально допустимое значение, определенное для типа `byte`. И наконец, при присваивании значения типа `byte` переменной типа `char` информация не теряется, поскольку используется приведение типа.

Минутный практикум

1. Что такое операция приведения типа?
2. Можно ли без выполнения операции приведения типа присвоить значение типа `short` переменной типа `int`? А значение типа `byte` — переменной типа `char`?

3. Запишите оператор `x = x + 23;` другим способом.

Приоритетность операторов

В табл. 2.3 показан порядок приоритетности всех операторов Cft от наивысшего к самому низкому. В таблицу включено несколько операторов, о которых будет рассказано позже.

Таблица 2.3. Приоритетность операторов в C#

Наивысший
`()` `[]` `.` `++(постфикс)` `--(постфикс)` `checked` `new` `sizeof` `typeof` `unchecked`
`!` `~` `(cast)` `+(унарный)` `-(унарный)` `++(префикс)` `--(префикс)`
`/` `%`
`+` `-`
`<<` `>>`
`<` `>` `<=` `>=` `is`
`==` `!=`
`&`
`^`
`|`
`&&`
`||`
`?:`
`=` `op=`
Самый низкий

Выражения

Операторы, переменные и литералы являются компонентами *выражений*. Выражением в C# является действительная комбинация этих компонентов. Если вы уже занимались программированием (или хотя бы изучали алгебру), то, вероятно, имеете представление о синтаксисе выражений. Однако при работе с ними нужно учитывать несколько важных моментов, о которых мы сейчас поговорим.

Преобразование типов в выражениях

В пределах выражения существует возможность совмещения двух и более различных типов данных при условии, что они являются совместимыми. Например, вы можете

1. Операция приведение типа — это явное преобразование типа данных.
2. Да. Нет.
3. Оператор можно записать как `x += 23;`.

совмещать в выражении данные типа `short` и `long`, поскольку оба эти типа являются числовыми. Если в выражении совмещаются различные типы данных, они преобразуются в один и тот же тип на основе алгоритма пошагового преобразования (то есть в соответствии с приоритетностью выполнения операций).

Преобразования выполняются посредством использования принятых в C# *правил автоматического преобразования типов в выражениях*. Ниже представлен алгоритм, определенный этими правилами для арифметических операций.

- Если один операнд имеет тип `decimal`, то второй операнд автоматически преобразуется к типу `decimal` (за исключением случаев, когда он имеет тип `float` или `double`; в этом случае произойдет ошибка).
- Если один из операндов имеет тип `double`, второй операнд автоматически преобразуется к типу `double`.
- Если один операнд имеет тип `float`, второй операнд автоматически преобразуется к типу `float`.
- Если один операнд имеет тип `ulong`, второй операнд автоматически преобразуется к типу `ulong` (за исключением случаев, когда он имеет тип `sbyte`, `short`, `int` или `long`; в этих случаях произойдет ошибка).
- Если один операнд имеет тип `long`, второй операнд автоматически преобразуется к типу `long`.
- Если один операнд имеет тип `uint`, а второй операнд - тип `sbyte`, `short` или `int`, то оба операнда автоматически преобразуются к типу `long`.
- Если один операнд имеет тип `uint`, второй операнд автоматически преобразуется к типу `uint`.
- Если ни одно из вышеуказанных правил не применялось, оба операнда преобразуются к типу `int`.

Обратите внимание на некоторые моменты, касающиеся правил автоматического преобразования типов. Не все типы данных могут совмещаться в выражениях (в частности, невозможно автоматическое преобразование данных типа `float` или `double` в тип `decimal` и невозможно совмещение данных типа `ulong` с любым другим типом знаковых целочисленных данных). Совмещение этих типов требует использования операции явного приведения типа.

Внимательно прочитайте последнее правило, в котором говорится, что если ни одно из вышеперечисленных правил не применялось, то все операнды преобразуются к типу `int`. Следовательно, все значения, имеющие тип `char`, `sbyte`, `byte`, `ushort` и `short`, в выражении преобразуются к типу `int` для выполнения вычислений. Такая процедура называется автоматическим преобразованием к целочисленному типу. Это также означает, что результат всех математических операций будет иметь тип, которому для хранения значения выделено не меньше битов, чем типу `int`.

Важно понимать, что автоматическое преобразование типов применяется к значениям (операндам) только тогда, когда выражение вычисляется. Например, если внутри выражения значение переменной типа `byte` преобразуется к типу `int`, то за пределами выражения переменная сохраняет тип `byte`.

Учтите, что автоматическое преобразование типа может привести к неожиданному результату. Например, когда арифметическая операция проводится с двумя значениями типа `byte`, выполняется такая последовательность действий: вначале операнды `byte` преобразуются к типу `int`, а затем вычисляется выражение, результат которого тоже будет иметь тип `int`. Результат операции над двумя значениями `byte`

будет иметь тип int. Это довольно неожиданное следствие выполнения вышеуказанного правила, поэтому программист должен контролировать тип переменной, которой будет присвоен результат. Применение правила автоматического преобразования к целочисленному типу рассматривается в следующей программе:

```
 // В программе демонстрируется применение правила автоматического
 // преобразования к целочисленному типу.

using System;

class PromDemo {
  public static void Main() {
    byte b;
    int i;

        b = 10;
        i = b * b; // В данном случае не требуется явное приведение типа.

    b = 10;
    b = (byte) (b * b); // Тип результата должен быть приведен
                        // к типу переменной b.

    Console.WriteLine("Значения переменных i и b: " + i + " " + b);
  }
}
```

Приведения типов не требуется, когда результат вычисления выражения `b * b` присваивается переменной `i`, так как переменная `b` автоматически преобразуется к типу `int` при вычислении выражения. Но если вы попытаетесь присвоить результат вычисления выражения `b * b` переменной `b`, необходимо будет выполнить операцию приведения типа обратно в тип `byte`! Вспомните это, если получите неожиданное сообщение об ошибке, возникшей из-за несовместимости типов в выражении.

Такая же ситуация возникает при проведении операций с данными типа `char`. Например, в следующем фрагменте кода необходимо выполнить приведение типа результата вычисления выражения обратно к типу `char`, поскольку в выражении переменные `ch1` и `ch2` преобразуются в тип `int`:

```
char ch1 = 'a', ch2 = 'b';

ch1 = (char) (ch1 + ch2);
```

Без приведения типа результат сложения значений переменных `ch1` и `ch2` имеет тип `int` и не может быть присвоен переменной типа `char`.

Приведение типов применяется не только для преобразования типов при присваивании. Например, в следующей программе для получения дробной части результата вычисления выражения результат приводится к типу `double`, поскольку без операции приведения он имел бы тип `int` и его дробная часть была бы утеряна.

```
// В программе демонстрируется применение операции приведения типа
// к результату вычисления выражения.

using System;

class UseCast {
```

```
public static void Main() {
  int i;

  for(i = 0; i < 5; i++) {
   Console.WriteLine("Целочисленный результат вычисления выражения "
                     +i+"/3: " + i / 3);
    Console.WriteLine("Результат вычисления выражения, выводимый" +
              "с дробной частью: {0:#.##}",(double) i / 3);
    Console.WriteLine();
```

Ниже показан результат выполнения этой программы:

```
Целочисленный результат вычисления выражения 0/3: 0
Результат вычисления выражения, выводимый с дробной частью:

Целочисленный результат вычисления выражения 1/3: 0
Результат вычисления выражения, выводимый с дробной частью: .33

Целочисленный результат вычисления выражения 2/3: 0
Результат вычисления выражения, выводимый с дробной частью: .67

Целочисленный результат вычисления выражения 3/3: 1
Результат вычисления выражения, выводимый с дробной частью: 1

Целочисленный результат вычисления выражения 4/3: 1
Результат вычисления выражения, выводимый с дробной частью: 1.33
```

Ответы профессионала

Вопрос. Происходит ли преобразование типов в выражениях с унарными операторами, такими как унарный минус?

Ответ. Да. Для унарных операций операнды, у которых диапазон допустимых значений меньший, чем у `int` (такие, как `byte`, `sbyte`, `short` и `ushort`), преобразуются к типу `int`. Операнд типа `char` также преобразуется к типу `int`. Кроме того, если операнду типа `uint` присваивается отрицательное значение, он преобразуется к типу `long`.

Использование пробелов и круглых скобок

Для улучшения читабельности программы выражения в C# могут содержать символы табуляции и пробелы. Например, следующие два выражения аналогичны, но второе гораздо легче для чтения:

```
x=10/y*(127/x);

x = 10 / y * (127/x);
```

Круглые скобки повышают приоритет операторов, содержащихся внутри них (скобки используются точно так же, как в алгебре). Использование дополнительных круглых скобок не приведет к ошибкам или замедлению вычисления выражения, поэтому их можно применять для определения точного порядка действий и, следовательно, для

улучшения читабельности программы. Например, очевидно, что второе выражение читается проще, чем первое:

```
x = y/3-34*temp+127;

x = (y/3) - (34*temp) + 127;
```

Проект 2-3. Вычисление суммы регулярных выплат по кредиту

RegPay.cs

Как уже говорилось, тип данных `decimal` особенно удобно использовать в денежных вычислениях. Предлагаемая программа вычисляет сумму регулярных выплат по кредиту (например, на покупку автомобиля). Принимая начальные данные: срок кредита, число платежей за год и величину процентов по кредиту, программа вычисляет размер одного платежа. Поскольку это финансовые вычисления, для представления данных имеет смысл использовать тип `decimal`. В этом проекте демонстрируется применение операций приведения типов, а также метода `Pow()` из библиотеки C#.

Для вычисления размера платежа используется следующая формула:

$$Payment = \frac{IntRate * (Principal / PayPerYear)}{1 - ((IntRate / PayPerYear) + 1)^{-PayPerYear * NumYears}}$$

где в переменной `IntRate` указывается процент выплат по кредиту, в переменной `Principal` содержится значение стартового баланса, в переменной `PayPerYear` указывается число платежей в год, а в переменной `NumYears` задается срок погашения кредита в годах.

Отметим, что в знаменателе формулы вы должны возвести в степень соответствующее значение. Для этого используется математический метод `Math.Pow()`. Вот как он вызывается:

```
result = Math.Pow(base, exp);
```

Метод `Math()` возвращает значение основания степени (`base`), возведенного в показатель степени (`exp`). Аргументы метода `Pow()` должны иметь тип `double`, и возвращаемое методом значение также будет иметь тип `double`. Это означает, что вам необходимо использовать приведение типов для преобразования типа `double` в тип `decimal`.

Пошаговая инструкция

1. Создайте новый файл и назовите его `RegPay.cs`.
2. Используйте в программе следующие переменные:

```
decimal Principal;  // Значение стартового баланса.
decimal IntRate;    // Процент выплат по кредиту, например 0.075.
decimal PayPerYear; // Количество платежей в год.
decimal NumYears;   // Срок погашения кредита.
decimal Payment;    // Размер платежа.
```

```
decimal numer, denom;     // Вспомогательные переменные.
double b, e;              // Основание и показатель степени
                          // для вызова метода Pow().
```

(Поскольку большая часть вычислений будет осуществляться с использованием данных типа `decimal`, большинство переменных имеет тип `decimal`). Обратите внимание, что за каждым объявлением переменной следует комментарий, объясняющий ее использование. Это помогает понять, какие функции выполняет каждая переменная. Хотя мы не включали такие комментарии в большинство коротких программ, приводимых в этой книге, нужно отметить, что с увеличением объема и сложности программы эти комментарии часто становятся необходимыми для понимания алгоритма.

3. Добавьте в программу строки кода, специфицирующие информацию о кредите. В программе заданы следующие данные: значение стартового баланса равно $10000, проценты по кредиту составляют 7.5%, число выплат в год — 12, а срок погашения кредита 5 лет.

```
Principal = 10000.00m;
IntRate = 0.075m;
PayPerYear = 12.0m;
NumYears = 5.0m;
```

4. Добавьте строки кода, в которых производятся финансовые вычисления:

```
numer = IntRate * Principal / PayPerYear;

e = (double) -(PayPerYear * NumYears);
b = (double) (IntRate / PayPerYear) + 1;

denom = 1 - (decimal) Math.Pow(b, e);

Payment = numer / denom;
```

(Обратите внимание, как нужно использовать приведение типов данных для передачи значений методу `Pow()` и преобразования возвращаемого значения. Помните, что в C# не существует автоматического преобразования между типами данных `decimal` и `double`.)

5. Завершите программу оператором, который выводит значение ежемесячной выплаты:

```
Console.WriteLine("Размер ежемесячной выплаты:{0:C}", Payment);
```

6. Ниже приведен полный текст программы `RegPay.cs`:

```
/*
   Проект 2-3

   Программа вычисляет размер ежемесячной выплаты по кредиту.

   Назовите этот файл RegPay.cs.
 */

using System;
```

```
class RegPay {
  public static void Main() {
    decimal Principal; // Значение стартового баланса.
      decimal IntRate; // Процент выплат по кредиту,
                              // например 0.075.
    decimal PayPerYear; // Количество платежей в год.
    decimal NumYears; // Срок погашения кредита.
    decimal Payment; // Размер платежа.
    decimal numer, denom; // Вспомогательные переменные.
    double b, e; // Основание и показатель степени
                              // для вызова метода Pow().

      Principal = 10000.00m;
      IntRate = 0.075m;
      PayPerYear = 12.0m;
      NumYears = 5.0m;

    numer = IntRate * Principal / PayPerYear;

         e = (double) -(PayPerYear * NumYears);
         b = (double) (IntRate / PayPerYear) + 1;

    denom = 1 - (decimal) Math.Pow(b, e);

    Payment = numer / denom;

    Console.WriteLine("Размер ежемесячной выплаты: {0:C}", Payment);
  }

}
```

В результате выполнения программы будет выведена следующая строка:

```
Размер ежемесячной выплаты: 200.38 грн.
```

Протестируйте эту программу, прежде чем перейти к изучению дальнейшего материала (введите различные значения суммы кредита, сроков погашения и процентов по кредиту).

Контрольные вопросы

1. Почему в C# строго специфицируются диапазоны допустимых значений и характеристики его простых типов?
2. Что представляет собой символьный тип в C# и чем он отличается от символьного типа, используемого в других языках программирования?
3. Справедливо ли утверждение, что переменная типа `bool` может хранить любое значение, поскольку любое ненулевое значение является истинным?
4. Используя одну строку кода и escape-последовательности, напишите оператор `WriteLine();`, в результате выполнения которого будут выведены следующие три строки:

```
Первая.
Вторая.
Третья.
```

5. Какая ошибка содержится в данном фрагменте кода?

```
fo r(i = 0; i < 10; i++) {
   int sum;

   sum = sum + i;

}
Console.WriteLine("Сумма равна: " + sum);
```

6. Объясните различие между постфиксной (`d++`) и префиксной (`++d`) формами оператора инкремента.
7. Напишите фрагмент кода, в котором для предотвращения ошибки деления на ноль использован быстрый оператор `AND`.
8. К какому типу преобразуются типы `byte` и `short` в выражении?
9. Какой из нижеприведенных типов не может совмещаться в выражении со значением типа `decimal`:

а) `float`
б) `int`
в) `uint`
г) `byte`

10. Когда необходимо применять приведение типов?
11. Напишите программу, которая находит все простые числа в диапазоне от 1 до 100.
12. Самостоятельно перепишите программу, предназначенную для вывода таблицы истинности (проект 2-2), таким образом, чтобы в ней вместо escape-последовательностей использовались копирующиеся строковые литералы с вложенными знаками табуляции и символами новой строки.

Управляющие операторы

- Ввод символов с клавиатуры
- Детальное рассмотрение операторов `if` и `for`
- Оператор `switch`
- Цикл `while`
- Использование цикла `do-while`
- Оператор `break`
- Использование оператора `continue`
- Оператор `goto`

В этой главе будут рассмотрены операторы, которые управляют процессом выполнения программы. Существуют три категории управляющих операторов — *операторы выбора* (`if` и `switch`), *итерационные операторы* (`for`, `while`, `do-while` и `foreach`) и *операторы перехода* (`break`, `continue`, `goto` и `return`). Здесь мы подробно расскажем обо всех перечисленных управляющих операторах, за исключением операторов `return` и `fereach`, которые будут описаны далее в этой книге.

Ввод символов с клавиатуры

Прежде чем перейти к изучению управляющих операторов, сделаем небольшое отступление и расскажем о том, каким образом в C# можно работать в интерактивном режиме. В предыдущих главах нашей книги рассматривались программы, только выводящие информацию на экран, теперь же в них будет предусмотрено и получение данных от пользователя.

Для чтения символов с клавиатуры необходимо вызвать метод `Console.Read()`, который ожидает ввода символа с клавиатуры, а затем возвращает результат. Результат возвращается как целочисленное значение, поэтому он должен быть преобразован в тип `char`, чтобы переменной был присвоен этот же тип. По умолчанию строка, вводимая с клавиатуры, помещается в буфер, и для того чтобы введенные символы были переданы программе, нужно нажать клавишу [Enter]. Представленная ниже программа демонстрирует, как считываются символы, введенные с клавиатуры.

```
// В программе демонстрируется считывание символов, вводимых с
// клавиатуры.

using System;

class KbIn {
  public static void Main() {
    char ch;

    Console.Write("Нажмите символьную клавишу, после чего нажмите"+
                  " клавишу ENTER: ");

    ch = (char) Console.Read(); // Считывание символа.

    Console.WriteLine("Вы ввели символ: " + ch);
```

Чтение символа, вводимого с клавиатуры.

Ниже приведен пример работы программы.

```
Нажмите символьную клавишу, затем нажмите клавишу ENTER:
Вы ввели символ p
```

Иногда у программистов вызывает недовольство то, что строка ввода метода `Read()` помещается в буфер, поскольку это немного усложняет работу. Ведь при нажатии клавиши [Enter] во входной поток вводится последовательность символов перехода на новую строку, и эти символы остаются в буфере ввода до тех пор, пока не будут считаны. Поэтому при работе с некоторыми приложениями (в этой главе вам встретится такой пример) действительно возникает неудобство — перед новым вводом приходится сначала удалять эти символы из буфера посредством их считывания.

Минутный практикум

1. Как считывается символ, введенный с клавиатуры?
2. Что означаем помещение строки в буфер?

Оператор if

Оператор `it` уже рассматривался в главе 1, теперь мы расскажем о нем подробнее. Синтаксис оператора `if` с его второй частью `else` выглядит следующим образом:

```
if (condition) statement;
 else statement;
```

Здесь при выполнении условия оператора `if` инициируется только один оператор, если же условие не выполняется, но также инициализируется только один оператор, относящийся к ключевому слову `else`. Для оператора `if` не обязательно указывать его вторую часть `else`. К оператору могут относиться также блоки операторов. Общий синтаксис оператора `if` с использованием блоков операторов выглядит следующим образом:

```
if (condition)

         последовательность операторов

else

    последовательность операторов
```

Если условное выражение (`condition`) принимает значение `true`, то выполняется последовательность операторов, соответствующих оператору `if`, если же условное выражение принимает значение `false`, то при наличии оператора `else` выполняется последовательность операторов, соответствующая части `else`. Обе части оператора, `if` и `else`, не могут выполняться одновременно. Условное выражение, управляющее оператором `if` должно возвращать результат типа `bool`.

Для демонстрации работы оператора `if` разработаем программу простои игры на угадывание, которая предназначена для маленьких детей. И первой версии игры программа просит игрока угадать букву в диапазоне от A до Z. Если игрок верно выбирает букву ни клавиатуре, программа выводит па экран сообщение `**Правильно**`.

Ниже приведен текст л ой программы.

```
// Простая игра "Угадай букву".

using System;


  partic stated Void Main() {
    char ch, _____________ 'k';
```

---

1. Для чтения символа необходимо вызвать метод `Console.read()`.
2. При вводе строка помешается в буфер, поэтому необходимо нажать клавишу [Enter], прежде чем введенные символы будут переданы программе.

```
    Console.WriteLine("Введите символ.");

    ch = (char) Console.Read(); //Считывает символ, введенный с клавиатуры.

    if (ch == answer) Console.WriteLine("** Правильно **");
  }
}
```

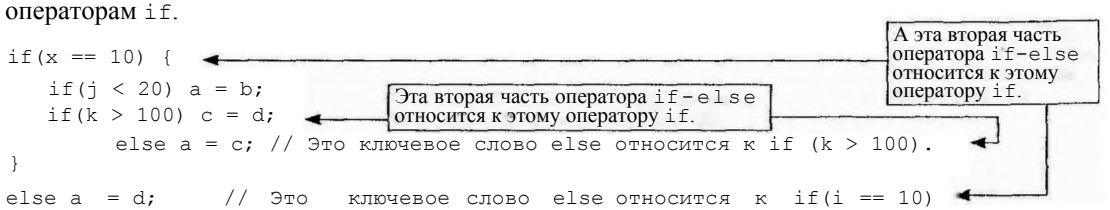
Программа просит ввести символ с клавиатуры и считывает его. Далее используется оператор `if` для проверки соответствия введенного символа значению переменной `answer`. в данном случае букве К. Если игрок вводит букву К, на экран выводится сообщение `** Правильно **`. При вводе должны использоваться символы верхнего регистра.

Игровую программу можно усовершенствовать. В следующей версии используется вторая часть оператора if-else для вывода сообщения о неверной попытке.

```
// Программа "Угадай букву" версия №2.

using System;

class Guess2 {
  public static void Main()  {
    char ch, answer = 'K';

    Console.WriteLine("Угадайте букву алфавита, \"спрятанную\" в программе.");
    Console.WriteLine("Введите символ.");

    ch = (char) Console.Read();// Считывает символ, введенный с клавиатуры.

    if(ch == answer) Console.WriteLine("** Правильно **");
    else Console.WriteLine("...Не угадали..");
  }
}
```

Вложенные операторы if

Вложенный оператор `if` используется для дальнейшей проверки данных после того, как условие предыдущего оператора `if` принимает -значение `true`. (То есть вложенный оператор применяется в тех случаях, когда для выполнения действия требуется соблюдение сразу нескольких условий, которые не могут быть указаны в одном условном выражении.) В программировании вложенные операторы `if` используются Достаточно часто. Необходимо помнить, что во вложенных операторах `if-else` вторая часть оператора (`else`) всегда относится к ближайшему оператору `if`, за Условным выражением которого следует оператор ; или блок операторов. В приведенном ниже примере указаны два ключевых слова `else`, относящихся к разным операторам `if`.



В комментарии указано, что последнее ключевое слово `else` не ассоциировано с оператором `if(j < 20)`, поскольку оно находится за блоком кода, относящимся к оператору `if(i == 10)`. То есть последнее ключевое слово `else` относится к оператору `if(i == 10)`, а не к оператору `if(j < 20)`, хотя этот оператор расположен ближе к ключевому слову.

Вложенный оператор `if` можно использовать для дальнейшего усовершенствования программы «Угадай букву». В новой версии обеспечивается обратная связь с игроком программа подсказывает, в каком направлении следует искать букву.

```
// Программа "Угадай букву" версия №3

using System;

class Guess3 {
    public static void Main() {
      char ch, answer — 'K';

     Console.WriteLine('Угадайте букву алфавита, \"спрятанную\" в программе.");
     Console.WriteLine("Введите символ. " ) ;

           ch =(char) Console.Read();// Считывание символа.

     if(ch == answer) Console.WriteLine ("*** Правильно ***");
     else {
       Console.Write("Попытайтесь поискать ");

       // Вложенный оператор if.
       if(ch < answer) Console.WriteLine ("выше по алфавиту.");
     else Console.WriteLine("ниже по алфавиту.”);
    }
   }
}
```

Вложенный оператор if.

Пример выполнения программы показан ниже.

```
Угадайте букву алфавита, "спрятанную" в программе.
Введите символ. В
Попытайтесь поискать выше в алфавите.
```

Цепочка операторов if-else-if

В программировании часто используется *цепочка* операторов `if-else-if` — конструкция, состоящая из вложенных операторов `if`. Выглядит она следующим образом:

```
if(condition)
    statement;
else if(condition)
    statement;
else if(condition)
    statement;
.
.
.
else
    statement;
```

условные выражения оцениваются сверху вниз. Как только найдено условие, принимающее значение `true`, выполняется ассоциированный с этим условием оператор (`statement`), а остальная часть цепочки пропускается. Если ни одно из условий не принимает значение `true`, то выполняется последний оператор `else`, который можно рассматривать как оператор по умолчанию. Если же последний оператор `else` отсутствует, а все условные выражения принимают значение `false`, то программа не выполняет никаких действий.

В следующей программе показано использование цепочки операторов `if-else-if`.

```
// В программе демонстрируется использование цепочки операторов if-else-if.

using System;

class Ladder {
  public static void Main() {
    int x;

    for (x=0; x<6; x++) {
      if (x==1)
        Console.WriteLine("Значение переменной x равно 1.");
      else if(x=2)
        Console.WriteLine("Значение переменной x равно 2.");
      else if(x==3)
        Console.WriteLine("Значение переменной x равно 3.");
      else if(x==4)
        Console.WriteLine("Значение переменной x равно 4.");
      else
        Console.WriteLine("Значение переменной x не входит в "+
                          "диапазон значений от 1 до 4.");
    }
  }
```

Этот оператор выполняется по умолчанию.

Программа выводит следующие строки:

```
Значение переменной x не входит в диапазон от 1 до 4.
Значение переменной x равно 1
Значение переменной x равно 2
Значение переменной x равно 3
Значение переменной x равно 4
Значение переменной x не входит в диапазон от 1 до 4.
```

Как видите, оператор `else` выполняется только тогда, когда ни одно из предшествующих условных выражении оператора `if` не принимает значение `true`.

Минутный практикум

1. Какого типа должен быть результат условного выражения оператора `if`?
2. С каким из операторов `if` всегда ассоциируется оператор `else`?
3. Что такое цепочка операторов `if-eise-if`?

1. Результат условного выражения оператора `if` должен иметь тип `bool`.
2. Оператор `else` всегда ассоциируется с ближайшим оператором `if`, после блока операторов которого (или единственного оператора) он находится.
3. Цепочка операторов `if-else-if` является последовательностью вложенных операторов `else-if`.

Оператор switch

Оператор switch является вторым оператором выбора в C#. Этот оператор позволяет программе выбрать нужное действие из перечня вариантов. Принципы работы вложенных операторов if-else и операторов switch различаются, причем во многих ситуациях использование оператора switch является более эффективным. Оператор switch работает следующим образом: значение выражения последовательно сравнивается с каждой константой из списка. При совпадении значения с одной из констант выполняется ассоциированная с ней последовательность операторов. Общая форма синтаксиса оператора switch следующая:

```
switch (expression) {
   case constant 1:
      statement sequence
      break;
   case constant 2:
      statement sequence
      break;
   case constant 3:
      statement sequence
      break;
.
.
.
   default:
      statemenc sequence
      break;
}
```

Выражение (expression) в операторе switch должно быть целочисленного типа (char, byte, short или int) либо типа string (который описан далее в этой книге) Выражения с плавающей точкой запрещены для использования в операторе switch Очень часто в качестве выражения, управляющего оператором switch используется просто переменная. Константы (constant) ветвей case оператора switch должны быть литералами и иметь тип, совместимый с выражением. Кроме того, в одном блоке оператора switch они не могут иметь одинаковые значения

Последовательность операторов (statement sequence), относящихся к ветви default, выполняется тогда, когда ни одна из констант ветвей case не соответствует выражению. Ветвь default не обязательна. Если при ее отсутствии в операторе switch ни одна из констант не соответствует выражению, то никакие действия не будут выполнены. Если соответствие обнаружено, то выполняется последовательность операторов, ассоциированная с этой ветвью case, затем с помощью оператора break контроль над выполнением программы передается конструкции, следующей за оператором switch.

Приведем пример программы, демонстрирующей применение оператора switch.

```
// В программе демонстрируется использование оператора switch

using System;

class SwitchDemo {
  public static void Main()    {
    int i;
```

```
      for(i-0;i<10;i++)
       switch(i)  {
         case 0:
           Console.WriteLine("Значение переменной i равно 0.");
           break;
         case 1:
           Console.WriteLine("Значение переменной i равно 1.");
           break;
         case 2:
           Console.WriteLine("Значение переменной i равно 2.");
           break;
         case 3:
           Console.WriteLine("Значение переменной i равно 3.");
           break;
         case 4:
           Console.WriteLine("Значение переменной i равно 4.");
           break;
         default:
           Console.WriteLine("Значение переменной i больше или равно 5.");
           break;
       }

  }
}
```

Ниже показан результат выполнения этой программы.

```
Значение переменной     i равно 0.
Значение переменной     i равно 1.
Значение переменной     i равно 2.
Значение переменной     i равно 3.
Значение переменной     i равно 4.
Значение переменной     i больше или  разно 5.
Значение переменной     i больше или  равно 5.
Значение переменной     i больше или  равно 5.
Значение переменной     i больше или  равно 5.
Значение переменной     i больше или  равно 5.
```

Как видите, каждый раз при прохождении цикла выполняется оператор, ассоциированный с ветвью `case`, константа которой совпала со значением переменной `i`. Все другие операторы пропускаются. Когда значение переменной `i` больше или равно 5, ни одна из констант не соответствует этому значению, то есть в этом случае выполняется последовательность операторов, определенных для ветви `default`.

В предыдущем примере оператор `switch` управлялся переменной типа `int`. Уже говорилось, что управлять оператором `switch` можно с помощью выражения любого целочисленного типа. Следующая программа демонстрирует использование выражения типа `char` и константы ветвей `case`, также имеющих тип `char`:

```
// В программе для управлением оператором switch используется выражение,
// имеющее тип char.

using System;

class SwitchDemo2 {
  public static void Main() {
    char ch;

    for (ch='A'; ch <= 'E' ;ch++)
```

```
      switch(ch) {
        case 'A':
          Console.WriteLine("Значением переменной ch является символ A.");
          break;
        case 'B':
          Console.WriteLine("Значением переменной ch является символ B.");
          break;
        case 'C':
          Console.WriteLine("Значением переменной ch является символ C.");
          break;
        case 'D':
          Console.WriteLine("Значением переменной ch является символ D.");
          break;
        case 'E':
          Console.WriteLine("Значением переменной ch является символ E.");
          break;
      }
  }
}
```

Результат выполнения этой программы показан ниже:

```
Значением переменной ch является символ A.
Значением переменной ch является символ B.
Значением переменной ch является символ C.
Значением переменной ch является символ D.
Значением переменной ch является символ E.
```

Обратите внимание, что в данной программе отсутствует ветвь `default`, которая не является обязательной и может быть пропущена, если в ней нет необходимости.

Последовательность операторов, ассоциированная с одной ветвью `case`, не должна передавать управление программой следующей ветви `case` (это называется правилом запрещения передачи управления программой между ветвями `case`), такое действие в C# рассматривается как ошибка. Во избежание этого последовательности операторов ветвей `case` заканчиваются оператором `break`. (Предотвратить передачу управления можно и другими способами, но чаще всего для этого используется именно оператор `break`.) Если оператор `break` расположен после последовательности операторов ветви `case`, это приводит к выходу из всего оператора `switch` и передаче управления программой следующей конструкции за пределами `switch`. Оператор `default` также должен следовать этому правилу, и поэтому обычно заканчивается оператором `break`.

Недопустимо, чтобы одна последовательность операторов ветви `case` передавала управление программой операторам другой ветви, но две и более ветви `case` могут быть связаны с одной и той же последовательностью кода, как показано в следующем примере:

```
switch(1)  (
  case Is
  case 2s
  case 3s Console.WriteLine("Переменная i имеет значение 1, 2 или 3");
    break;
  case 4s Console.WriteLine("Значение переменной i равно 4");
    break;
}
```

Эти ветви `case` вызывают один и тот же оператор.

Если переменная `i` имеет значение 1, 2 или 3, то выполняется первый оператор `WriteLine();`. Если значение переменной равно 4, то выполняется второй оператор `WriteLine();`. Возможность использования одной последовательности операторов несколькими ветвями `case` не нарушает правило, запрещающее передачу управления

программой между ветвями case, и весьма часто используется в программировании. Так, в следующей программе она используется для разделения букв алфавита на гласные и согласные. (Для корректной работы программы следует вводить символы нижнего регистра.)

```
// В программе демонстрируется использование несколькими ветвями case одной
// последовательности операторов для разделения бука на гласные и согласные.

using System;

class VowelsAndConsonants {
  public static void Main() {
    char ch;

    Console.Write("Введите букву: ");
    ch = (char) Console.Read();
    switch(ch) {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
        case 'y':
          Console.WriteLine("Это гласная буква.");
          break;
        default:
          Console.WriteLine("Это согласная буква.");
          break;
    }
  }
}
```

Если бы этот пример был написан без использования вышеназванной технологии, то запись одного и того же оператора WriteLine(); необходимо было бы повторить шесть раз.

Ответы профессионала

Вопрос. В каких случаях вместо оператора switch нужно использовать цепочку операторов if-else-if?

Ответ. Общее правило таково: цепочка операторов if-else-if используется в тех случаях, когда выражение условия может иметь много значений (то есть когда в выражении условия удобнее применить оператор сравнения if(j < 0)) или требуется сравнение нескольких переменных. В качестве примера рассмотрим следующую цепочку операторов:

```
if(x < 10)      // ...
else if(y != 0)// ...
else if(ldone) // ...
```

Такую последовательность нельзя записать с помощью оператора switch, поскольку все три условия основаны на различных переменных и различных типах. Также цепочка операторов if-else-if используется для проверки значений с плавающей точкой или для проверки объектов других типов, которые нельзя применять в выражении оператора switch.

Вложенный оператор switch

Оператор switch может быть частью внешнего оператора switch. Внутренний оператор switch называется вложенным. Константы ветвей case внутреннего и внешнего операторов switch могут иметь одинаковые значения, и это не вызовет конфликта при выполнении программы. Например, следующий фрагмент кода вполне допустим:

```
switch(ch1) {
  case 'A': Console.WriteLine("Эта константа 'A' является частью "+
                              " внешнего оператора switch.");
    switch(ch2) {
      case 'A':
        Console.WriteLine("Эта константа 'A' является частью "+
                          "внутреннего оператора switch.");
        break;
      case 'B': // ...
    } // Закрывающая скобка внутреннего оператора switch.
    break;
  case 'B':  // ...
```

Ответы профессионала

Вопрос. В языках C, C++ и Java последовательность операторов одной ветви case может передавать управление программой последовательности операторов следующей ветви case. Почему это не разрешено с C#?

Ответ. В C# правило запрещения передачи управления программой между ветвями case принято по двум причинам. Во-первых, это правило позволяет компилятору организовать порядок перечисления операторов ветвей case наиболее оптимальным образом. Это было бы невозможно, если бы последовательность операторов одной ветви case могла передавать управление следующей последовательности операторов. Во-вторых, в C# каждая последовательность операторов должна заканчиваться оператором break, что также не позволяет передавать управление программой между ветвями case.

Минутный практикум

1. Какого типа должно быть выражение, управляющее оператором switch?
2. Что происходит, когда выражение оператора switch совпадает со значением одной из констант ветви case?
3. Может ли последовательность операторов, ассоциированная с одной ветвью case, передавать управление программой следующей ветви case?

---

1. Выражение оператора switch должно быть целочисленного типа, либо типа string.
2. При совпадении проверяемого значения со значением константы выполняется последовательность операторов, ассоциированная с этой ветвью case.
3. Нет, поскольку с C# для оператора switch существует правило, запрещающее передачу управления между ветвями case.

Проект 3-1. Построение простой справочной системы C#

Help.cs

В этом проекте мы построим простую справочную систему, которая выводит общую форму синтаксиса для управляющих операторов C#.

Программа выводит меню, содержащее список управляющих операторов, и ожидает, пока вы выберете один из них. После того, как выбор сделан, на экран выводится общая форма синтаксиса этого оператора. В первой версии программы будет доступна только информация о синтаксисе операторов `if` и `switch`. Общие формы синтаксиса других управляющих операторов будут добавлены в следующих проектах.

Пошаговая инструкция

1. Создайте файл и назовите его `Help.cs`.
2. Программа начинает работу с вывода на экран следующего меню:

```
Справка по синтаксису:
    1. Оператор if.
    2. Оператор switch.
Введите порядковый номер оператора:
```

поэтому вам необходимо использовать такую последовательность операторов:

```
Console.WriteLine("Справка по синтаксису:");
Console.WriteLine("   1. Оператор if.");
Console.WriteLine ("  2. Оператор switch.”);
Console.Write("Введите порядковый номер оператора:     ");
```

3. Программа получает информацию о выборе пользователя, вызывая метод `Concole.Read()`, как показано ниже:

```
choice = (char) Console.Read();
```

4. После получения информации о выборе пользователя программа использует оператор `switch`, представленный ниже, чтобы вывести на экран синтаксис выбранного оператора.

```
switch(choic e) {
  case '1':
    Console.WriteLine("Синтаксис оператора if:\n");
    Console.WriteLine("if(condition) statement;");
    Console.WriteLine("else statement;");
    break;
  case '2':
    Console.WriteLine("Синтаксис оператора switch:\n");
    Console.WriteLine("switch(expression) {");
    Console.WriteLine(" case constant:");
    Console.WriteLine("       statement  sequence");
    Console.WriteLine("     break;”);
    Console.WriteLine(" // ...”);
    Console.WriteLine("}");

    break;
  default;
    Console.Write("Порядковый номер указан неверно.");
    break;
}
```

Отмстим, что ветвь default перехватывает информацию о неправильно указанных порядковых номерах. Например, если пользователь введет значение 3, не совпадающее ни с одной из копстат оператора с.-se, то эго приведет к выполнению оператора, определенного для ветви Jo fault..

Ниже представлен полный листинг программы Heip.cs:

```
/*         Проект 3-1
   Простая справочная система.
 */
using System;

class Help {
  public static void Main() {
    char choice;

    Console.WriteLine("Справка по синтаксису:");
    Console.WriteLine(" 1. Оператор if.");
    Console.WriteLine (" 2. Оператор switch.");
    Console.Write("Введите порядковый номер оператора: ");
    choice = (char) Console.Read();

    Console-WnteLine ("\n") ;

    switch(choice) {

    case '1':
      Console.WriteLine("Синтаксис оператора if:\n");
      Console.WriteLine("if (condition) statement;");
      Console.WriteLine("else    statement;");
      break;
    case '2':
      Console.WriteLine("Синтаксис оператора switch:\n");
      Console.WriteLine("switch(expression) ;") ;
      Console.WriteLine(" case constant:");
      Console.WriteLine("      statement sequence ");
      Console.WriteLine("      break;");
      Console.WriteLine("    // ...");
      Console.WriteLine("}")  ;
      break;
    default:
      Console.Write("Порядковый  номер  указан  неверно.");
      break;
  }
 }
}
```

Ниже приведен пример выполнения этой программы:

```
Справка по синтаксису:
   1. Оператор if.
   2. Оператор switch.
Введите порядковый номер оператора:   1

Синтаксис  оператора  if:
if (condition) statement;
else statement;
```

Цикл for

С первой главы этой книги в программах используется простейшая форма цикла for. Теперь мы расскажем обо всех возможностях этого оператора, и вы увидите его мощь и гибкость. Начнем с рассмотрения основ цикла for и его наиболее традиционных форм.

Общий синтаксис цикла for для повторения одного оператора выглядит так:

```
for(initialization; condition; iteration) statement;
```

Для повторения блока операторов используется цикл, запись которого отличается от предыдущей только наличием открывающей и закрывающей скобок, между которыми и находится блок выполняемых операторов.

```
for(initialization; condition; iteration)

   statement sequence
```

При *инициализации управляющей переменной цикла* (initialization — первая составляющая цикла), которая действует как счетчик для управления циклом, ей присваивается начальное значение. Для этого обычно используется оператор присваивания. Составляющая цикла condition — это выражение условия, имеющее тип bool, от истинности которого (при проверке измененного значения управляющей переменной) зависит, будет ли выполнен еще один проход цикла. Составляющая цикла iteration — это выражение, определяющее величину (шаг), на которую будет изменяться значение контрольной переменной при каждом повторении цикла. Отметим, что эти три основных элемента цикла должны быть разделены точкой с запятой. Цикл for повторяется до тех пор, пока в результате проверки выражения условия будет возвращаться значение true. Если будет возвращено значение false, цикл завершится и управление программой будет передано оператору, следующему за циклом for.

В приведенной ниже программе цикл for используется для вывода результатов вычисления квадратных корней чисел от 1 до 99. Программа также выводит ошибки округления для каждого результата.

```
// Программа выводит результаты вычисления квадратных корней чисел
// от 1 до 99.

using System;

class SqrRoot {
   public static void Main() {
     double num, sroot, rerr;

    for(num = 1.0;num < 100.0; num++)  {
      sroot = Math.Sqrt(num);
      Console.WriteLine("Квадратный корень числа " + num + " равен " + sroot);

      // Вычисление ошибки округления,
      rerr = num - (sroot * sroot);
      Console.WriteLine("Ошибка округления равна " + rerr);
       Console.WriteLine();
    }
  }
}
```

Обратите внимание, что ошибка округления вычисляется следующим образом: возводится в квадрат значение квадратного корня числа, затем этот результат отнимается от первоначального значения. Следовательно, значение разности и будет составлять ошибку округления. Иногда ошибка округления будет происходить еще и при возведении в квадрат значения квадратного корня, то есть будет округляться сама ошибка округления. Этот пример показывает, что если в выражениях используются числа с плавающей точкой, то эти вычисления могут быть недостаточно точными.

Переменная цикла может не только увеличивать, но и уменьшать свое значение, кроме того, контрольная переменная может изменяться на любую величину при каждом проходе цикла. Например, следующая программа выводит числа в диапазоне от 100 до -100 с шагом уменьшения переменной цикла, равным 5:

```
// Цикл с отрицательным шагом изменения переменной цикла.

using System;

class DecrFor {
  public static void Main()    {
    int x;

    for(x = 100;x > —100;x — 5)
      Console.WriteLine(x);
  }
}
```

Переменная цикла уменьшается на 5 при каждом прохождении цикла.

Следует отмстить, что в цикле `for` выражение условия всегда проверяется в начале цикла. Это означает, что если изначально выражение условия принимает значение false, код внутри цикла не будет выполнен ни разу. Например:

```
for(count = 10);count < 5;count++)
   x += count;// Этот оператор выполняться не будет.
```

Оператор этого цикла не будет выполняться никогда, поскольку его управляющая переменная `count` имеет значение больше 5 в начале цикла, что изначально делает условное выражение `count < 5` ложным.

Некоторые варианты цикла for

Цикл `for` является одним из самых разносторонних операторов. Например, в нем могут использоваться несколько управляющих переменных цикла. Рассмотрим следующую программу:

```
// Использование запятых в управляющих частях цикла for.

using System;

class Comma {
  public static void Mam() {
    int i, j;

    for(i=0, j=10; i < j; i+-, j--)
      Console.WriteLine("Значения переменных i и j: " + i + " " + j);
  }
}
```

Обратите внимание, что цикл имеет две управляющие переменные.

Ниже показан результат выполнения этой программы.

```
Значения переменных i и j 0 10
Значения переменных i и j 1 9
Значения переменных i и j 2 8
Значения переменных i и j 3 7
Значения переменных i и j 4 6
```

Здесь два оператора инициализации и два итерационных выражения разделены запятыми. Когда цикл начинает свою работу, обеим переменным присваиваются начальные значения. Каждый раз при прохождении цикла значение переменной `i` увеличивается на единицу, а значение переменной `j` уменьшается на единицу. Управление циклом `for` с помощью нескольких переменных часто бывает удобным и упрощает алгоритмы, реализуемые с помощью этого цикла. В цикле вы можете использовать любое число операторов инициализации и итерационных операторов, но на практике применение более двух переменных для контроля цикла `for` делает его слишком громоздким.

Условие, управляющее циклом, может быть любым действительным выражением, которое принимает значение типа `bool`. Условие не обязательно должно включать управляющую переменную цикла. В следующем примере цикл продолжает выполняться до тех пор, пока пользователь не введет символ `S`.

```
// Цикл выполняется до тех пор, пока не будет введен символ S.

using System;

class ForTest {
  public static void Main()    {
    inc i;

    Console.WriteLine("Для выхода из программы нажмите клавишу S.");

    for(i = 0;(char) Console.Read() != 'S';i++)
      Console.WriteLine("Проход № " + i);
  }

}
```

Недостающие части цикла for

В C# можно создавать интересные варианты цикла `for`, оставляя пустыми все или некоторые части инициализации, условия и итерации. Например, рассмотрим следующую программу:

```
// Некоторые определяющие части цикла for могут быть пустыми.

using System;

class Empty {
  public static void Main() {
    int i, count = 0;

    for(i = 0;i < 10;)  {          <-- Итерационное выражение пропущено.
     count++;
     Console.WriteLine("Проход цикла № " + count);
      i++;// Увеличение управляющей переменной цикла ка единицу.
    }
  }
}
```

Здесь итерационное выражение цикла for пропущено. Вместо этого контрольная переменная цикла i увеличивается на единицу внутри тела цикла. Это означает, что каждый раз при выполнении цикла выполняется проверка значения переменной i (значение переменной сравнивается со значением 10), но дальнейших действий не происходит. (Переменная count введена только для того, чтобы счет проходов цикла начинался с цифры 1, а не 0.) Поскольку изменение переменной 1 происходит внутри тела цикла, он работает нормально, выводя на экран следующую информацию:

```
Проход цикла № 1
Проход цикла № 2
Проход цикла № 3
Проход цикла № 4
Проход цикла № 5
Проход цикла № 6
Проход цикла № 7
Проход цикла № 8
Проход цикла № 9
Проход цикла № 10
```

В следующем примере инициализирующее выражение также вынесено за пределы определяющей части цикла for.

```
// В этой программе инициализирующее выражение также вынесено за
// пределы определяющей части цикла for.

using System;

class Empty2 {
  public static void Main() {
    int i, count = 0;

    i = 0;  // Инициализирующее выражение находится вне цикла for.
    for(; i < 10;) {
      count
      Console.WriteLine("Проход № " + count);
      i++;// Увеличение управляющей переменной цикла на единицу.
    }
  }
}
```

Инициализирующее выражение **вынесено** за пределы цикла.

В данной версии программы инициализация переменной i происходит до начала цикла, и эго выражение не является частью цикла. Однако чаще контрольная переменная цикла инициализируется внутри определяющей части цикла for. Обычно операция инициализации выносится за пределы цикла только в тех случаях, когда начальное значение управляющей переменной является результатом вычисления сложного выражения, которое нельзя разместить внутри оператора for.

Бесконечный цикл

Используя оператор for с пустым условным выражением, вы можете создать бесконечный цикл. Например, в следующем фрагменте кода показан способ создания бесконечного цикла:

```
for(;;)  //  Бесконечный цикл.
{
  // ...
}
```

Такой цикл будет работать вечно. Хотя для выполнения некоторых залам программирования (например, для реализации командных процессоров операционной системы) необходимы бесконечные циклы, существуют специальные средства, при помощи которых можно прекратить выполнение большинства таких циклов. В конце этой главы мы расскажем, как можно остановить бесконечный цикл. (Подсказка: это выполняется с помощью оператора `break`.)

Циклы, не имеющие тела

В C# тело цикла `for` (или любого другого цикла) может быть пустым. Это возможно, потому что *нулевой оператор* синтаксически является действительным. Циклы без села используются достаточно часто. Например, в следующей программе такой цикл применяется для суммирования чисел от 1 до 5.

```
// Тело цикла может быть пустым.

using System;

class Empty3 {
  public static void Main()   {
    int i;
    int sum = 0;

    // Вычисляется сумма чисел от 1 до 5.
    for(i = l;i <= 5;sum +— i++);          В этом цикле тело отсугствуст!

    Console.WriteLine("Сумма равна: " + sum);
  }
}
```

Ниже показан результат выполнения этой программы.

```
Сумма равна 15
```

Отметим, что вычисление суммы производится полностью в пределах оператора `for`, следовательно, в теле оператора нет необходимости. Особое внимание обратите на итерационное выражение

```
sum += i++
```

Такие операторы выглядят несколько непривычно, тем не менее, они часто используются в профессионально написанных C#-программах. Описать работу приведенного выше оператора можно следующим образом: «добавить к переменной `sum` значение переменной `sum` плюс значение переменной `i`, затем увеличить значение переменной `i` на единицу». То есть предыдущий оператор аналогичен последовательности операторов

```
sum = sum + i;
i++;
```

Объявление управляющих переменных цикла внутри цикла for

Очень часто управляющая циклом `for` переменная необходима только для самого цикла и больше нигде не используется. В таких случаях можно объявлять переменную

внутри инициализирующей части цикла. Например, представленная ниже программа вычисляет как сумму, так и факториал чисел от 1 до 5. В этой программе управляющая переменная `i` объявляется внутри цикла `for`.

```
// В программе управляющая переменная объявляется внутри
// инициализационной части цикла for.

using System;

class ForVar {
  public static void Main() {
    int sum = 0;
    int fact = 1;

    // Вычисление суммы и факториала чисел от 1 до 5.
    for(int i *- l;i o 5;i++)  !
      sum  +=  i;// Областью видимости переменной i является весь цикл.
      fact *= i;
    }

    // В этом месте программы переменная i не видна.

    Console.WriteLine("Сумма чисел от 1 до 5 = " + sum);
    Console.WriteLine("Факториал числа 5 = " + fact);
  }
}
```

Переменная ***i*** объявляется внутри оператора for.

При объявлении переменной внутри цикла `for` ее область видимости ограничена рамками цикла (то есть область видимости переменной ограничена циклом `for`), за его пределами переменная прекращает свое существование. Следовательно, в предыдущем примере переменная `i` недоступна за пределами цикла. Если вам необходимо использовать управляющую переменную цикла в другом месте программы, то объявите эту переменную за пределами цикла `for`.

Прежде чем перейти к изучению следующей темы, потренируйтесь в использовании различных вариантов цикла `for`. Вы увидите, что он имеет огромные возможности.

Минутный практикум

1. Могут ли части оператора `for` быть пустыми?
2. Напишите синтаксис бесконечного цикла `for`.
3. Где заканчивается область видимости переменной, объявленной в пределах оператора `for`?

---

1. Да, все три части цикла `for` — инициализирующая, часть условия и итерационная — могут быть пустыми.

2. `for(;;)`.

3. Область видимости переменной, которая объявлена в пределах оператора `for`, ограничена рамками цикла. За пределами цикла эта переменная неизвестна.

Цикл while

Цикл while также широко применяется в C#-программах. Его синтаксис выглядит следующим образом:

```
while (condition) statement;
```

Здесь statement может быть одиночным оператором или блоком операторов, а condition является условием, которое управляет циклом, и может быть любым действительным булевым выражением. Оператор выполняется в том случае, если условие принимает значение true. Если же условие принимает значение false, управление программой передается строке кода, которая следует сразу за циклом.

Приведем пример программы, в которой цикл while используется для вывода на экран символов латинского алфавита.

```
// В программе демонстрируется использование цикла while.

using System;

class WhileDemo {
  public static void Main()  {
    char ch;

    // Использование цикла while для вывода на экран символов
    // латинского алфавита.
    ch = 'a';
    while(ch <= 'z')  {
      Console.Write(ch+  "  ");
      ch++;
    }
  }
}
```

Здесь символ a присваивается переменной ch в качестве начального значения. Каждый раз при прохождении цикла значение переменной ch выводится на экран, а затем увеличивается на единицу. Этот процесс продолжается до тех пор, пока значение переменной ch не превысит z.

Как и цикл for, цикл while проверяет условие в начале цикла, и это означает, что код цикла может вообще не выполняться. В приведенной ниже программе демонстрируется использование цикла while для вычисления целочисленной степени числа 2.

```
// В программе используется цикл while для вычисления целочисленной
// степени числа 2.

using System;

class Power {
  public static void Main() {
    int e;
    int result;

    for(int i=0;i < 10;i++) {
      result = 1;
      e = i;
      while(e > 0) {
        result *= 2;
```

```
        e--;
      }

      Console.WriteLine("2 в " + i + "-й степени - " + result);
    }
  }
}
```

Результат выполнения этой программы показан ниже.

```
2 в  0-й степени = 1
2 в  1-й степени = 2
2 в  2-й степени = 4
2 в  3-й степени = 8
2 в  4-й степени = 16
2 в  5-й степени = 32
2 в  6-й степени = 64
2 в  7-й степени = 128
2 в  8-й степени = 256
2 в  9-й степени = 512
```

Заметьте, что цикл `while` выполняется только в том случае, если значение переменной `e` больше 0. Поскольку при первом выполнении цикла `for` значение переменной `i`, а значит и переменной `e`, будет равно 0, цикл `while` выполняться не будет.

Цикл `do-while`

В завершение этой темы рассмотрим цикл `do-wile`. В отличие от циклов `for` и `while`, где условие проверяется в самом начале, в цикле `do-while` условие проверяется в конце цикла. Это означает, что цикл `do-while` всегда выполняется как минимум один раз. Синтаксис цикла `do-while` следующий:

```
do {
   statements;
} while (condition);
```

При наличии в цикле только одного оператора фигурные скобки не обязательны, но их часто используют для улучшения читабельности конструкции цикла `do-while`, поскольку его легко спутать с циклом `while`. Если при проверке условие принимает значение true, цикл `do-while` выполняется еще раз.

В следующей программе цикл продолжается до тех пор, пока пользователь не введет символ `q`.

```
// В программе демонстрируется использование цикла do-while.

using System;

class DWDemo {
  public static void Main() {
    char ch;

    do {
      Console.Write("Введите символ, после чего нажмите
                    " клавишу ENTER: ");
      ch = (char) Console.Read();//Считывание символа.
    } while{ch != 'q');
  }
}
```

Используя цикл do-while, мы можем усовершенствовать разработанную в начале главы программу «Угадай букву». Теперь программа будет выполняться до тех пор, пока буква не будет угадана.

```
   // Программа "Угадай букву" версия №4 .

      using System;

class Guess4 {
  public static void Main() {
    char ch, answer = 'K';

    do {
      Console.WriteLine("Угадайте букву алфавита, \"спрятанную”
                        "в программе.");
      Console.WriteLine("Введите символ: ");

      // В цикле считываемся символ, но игнорируются
      //  escape-последовательности \n и \r, соответствующие клавише ENTER,
      do {
        ch = (char) Console.Read();// Считывание символа.
      } while (ch == '\n' | ch == '\r');

      if (ch == answer) Console.WriteLine("** Правильно **");
      else {
        Console.Write("Попытайтесь поискать ");
        if(ch < answer) Console.WriteLine("выше в алфавите.");
        else Console.WriteLine("ниже в алфавите.\n");
      }
    } while(answer != ch);
  }
}
```

Ниже показан один из возможных результатов работы программы:

```
Угадайте букву алфавита, "спрятанную" в программе.
Введите символ: A
Попытайтесь поискать выше в алфавите.
Угадайте букву алфавита, "спрятанную” в программе.
Введите символ: Z
Попытайтесь поискать ниже в алфавите.
Угадайте букву алфавита, "спрятанную" в программе.
Введите символ: K
   ** Правильно **
```

Обратите внимание на интересную особенность этой программы. Представленный ниже цикл do-while считывает вводимый символ, пропуская символ возврата каретки или новой строки, которые могут быть во входном потоке.

```
// В цикле считывается символ, но игнорируются
  //  escape-послеловательности \n и \r, соответствующие клавише [Enter]
do {
  ch = (char) Console.Read();// Считывание символа.
} While(ch == '\n' | ch == '\r');
```

Этот цикл был введен в программу именно для того, чтобы она игнорировала ненужные символы. Как уже говорилось ранее, при вводе данных с клавиатуры строка ввода помещается в буфер, поэтому прежде чем символы будет переданы программе, необходимо нажать клавишу [Enter], При этом генерируются символы возврата

каретки и новой строки, которые остаются во входном буфере. Цикл `do-while` отбрасывает эти символы и продолжает считывание до тех пор, пока не встретится какой-либо другой символ.

Ответы профессионала

Вопрос. Циклы обладают очень большой гибкостью. Как правильно выбрать цикл для определенной задачи?

Ответ. Используйте цикл `for`. когда известно число повторений цикла, и цикл `do-while`, когда необходимо хотя бы одно прохождение цикла. Применение цикла `while` наиболее эффективно в тех случаях, когда число повторений цикла неизвестно.

Минутный практикум

1. Каково основное различие между циклами `while` и `do-while`?
2. Справедливо ли утверждение, что условие, управляющее циклом `while`, может быть любого типа?

Проект 3-2. Совершенствование справочной системы C#

`Help2.cs`

В этом проекте расширяется справочная система C#, созданная в проекте 3-1, — добавляется синтаксис циклов `for`, `while` и `do-while`. Программа также проверяет правильность выбора пользователя, продолжая выполнение цикла до тех пор, пока не будет введен корректный символ.

Пошаговая инструкция

1. Скопируйте код файла `Help.cs` и новый файл под названием `He1p2.cs`.
2. Измените фрагмент программы, выводящей на экран меню выбора, так, чтобы использовался представленный ниже цикл.

```
do {
  Console.WriteLine("Справка по синтаксису: ");
  Console.WriteLine(" 1. Оператор if");
  Console.WriteLine(" 2. Оператор switch")r
  Console.WriteLine(" 3. Цикл for");
  Console.WriteLine(" 4. Цикл while");
  Console.WriteLine(" 5. Цикл do-while\n");
  Console.Write("Введите порядковый номер оператора или цикла: ");
  do {
    choice = (char) Console.Read();
  } while(choice == '\n' | choice == '\r');
} while( choice < '1' | choice > '5');
```

---

1. Цикл `while` проверяет свое условие в начале цикла. Цикл `do-while` проверяет свое условие в конце цикла. Следовательно, цикл `do-while` всегда выполняется как минимум один раз.

2. Нет. Условие должно иметь тип `bool`.

Здесь, как и в предыдущей программе, никл `do-while` используется для того, чтобы игнорировать символы возврата каретки и новой строки, которые могут присутствовать во входном потоке. После произведенных изменений программа будет выполнять цикл и выводить на экран меню до тех пор, пока пользователь не введет корректный символ — цифру из диапазона от I до 5.

3. Расширьте оператор `switch`, включив в него синтаксис циклов `for`, `while` и `do-while`, как это показано ниже.

```
switch(choice)   {
  case '1':
    Console.WriteLine("Оператор if:\n");
    Console.WriteLine("if(condition) statement;");
    Console.WriteLine("else     statement;");
    break;
  case '2':
    Console.WriteLine("Оператор switch:\n") ;
    Console.WriteLine("switch(expression) (") ;
    Console.WriteLine(" case constant:");
    Console.WriteLine("       statement sequence");
    Console.WriteLine("       break;”);
    Console.WriteLine("     // ...");
    Console.WriteLine("}");
    break;
  case ’3':
    Console.WriteLine("Цикл for:\n");
    Console.Write("for(init;condition;iteration)” ) ;
    Console.WriteLine("     statement;");
    break;
  case '4':
    Console.WriteLine("Цикл while:\n");
    Console.WriteLine("while(condition)       statement;");
    break;
  case '5':
    Console.WriteLine("Цикл do-while:\n");
    Console.WriteLine("do {");
    Console.WriteLine(" statement;");
    Console.WriteLine("}    while    (condition);");
    break;
}
```

Отметим, что в этой версии оператора `switch` отсутствует ветвь `default`. Цикл, предназначенный для отображения меню, гарантирует ввод пользователем действительного символа, поэтому ветвь `default` для обработки некорректного пользовательского ввода не нужна.

4. Ниже представлен полный листинг программы `Help2.cs`:

```
/* Проект 3-2.
   Улучшенная  справочная  система,  использующая  для
   обработки пользовательского ввода цикл while.
*/

using System;

class Help2 (
  public static void Main()     {
    char choice;

    do {
```

```
        Console.WriteLine("Справка по синтаксису: ");
        Console.WriteLine(" 1. Оператор if");
        Console-WriteLine(" 2. Оператор switch");
        Console.WriteLine(" 3. Цикл for");
        Console.WriteLine(" 4. Цикл while");
        Console.WriteLine(" 5. Цикл do-while\n");
        Console.Write("Введите порядковый номер оператора или цикла: ");
       do {
          choice - (char) Console.Read();
        } while(choice == '\n' | choice == '\r');
      } while( choice < '1' | choice > '5');

      Console.WriteLine("\n");

      switch(choice) {
        case '1' :
          Console.WriteLine("Оператор if:\n");
          Console.WriteLine("if(condition) statement;");
          Console.WriteLine("else statement;");
          break;
        case '2':
          Console.WriteLine("Оператор switch;\n");
          Console.WriteLine("switch(expression) {");
          Console.WriteLine(" case constant:");
          Console.WriteLine(" statement sequence");
          Console.WriteLine(" break;");
          Console.WriteLine("    // ...");
          Console.WriteLine("}") ;
          break;
        case '3';
          Console.WriteLine("Цикл for:\n");
          Console.Write("for(init;condition;iteration)");
          Console.WriteLine(" statement;");
          break;
        case '4':
          Console.WriteLine("Цикл while:\n");
          Console.WriteLine("while(condition) statement;");
          break;
        case '5':
          Console.WriteLine("Цикл do-while:\n");
          Console.WriteLine("do {");
          Console.WriteLine(" statement;");
          Console.WriteLine("} while (condition);");
          break;
      }
    }
}
```

Использование оператора break для выхода из цикла

Использование оператора `break` предоставляет возможность принудительного и немедленного выхода из цикла без выполнения оставшегося кода в теле цикла и проверки условия цикла. То есть цикл завершает свою работу, когда оператор `break`

встречается внутри цикла, при этом управление программой передастся оператору, следующему за циклом. Приведем простой пример.

```
//  В программе демонстрируется использование оператора break
// дли выхода из цикла.
      using System;

ciass BreakDemo {
  public static void Main()   {
    int num;

    num = 100;

       // Цикл выполняется до тех пор, пока квадрат значения
    //  переменной i не превысит значение переменной num.
    lor (int i=0;i < num;i++) {

      if(i*i >= num) break;// Если выражение i*i >= 100 примет
                           //  значение true, будет выполнен оператор
                           // break, что приведет к выходу из никла.
      Console.Write (i + " ");
    }
    Console.WriteLine("Цикл завершен.");
   }
 }
```

Использование оператора `break` для выхода из цикла.

Эта программа выведет следующую строку:

```
0 1 2 3 4 5 6 7 8 9 Цикл завершен.
```

Цикл `for` должен повториться 100 раз, но выполнение оператора `break` приводит к досрочному прекращению работы цикла. Это происходит, когда значение переменной `i`, возведенное в квадрат, становится большим или равным значению переменной `num`.

Оператор `break` может использоваться в любом цикле, включая преднамеренно определенные программистом бесконечные циклы. Следующая программа просто считывает вводимые пользователем символы до тех пор, пока не будет нажата клавиша `q`.

```
// Программа считывает вводимые символы до тех пор,
// пока не будет нажата клавиша q.

using System;

class Break2 {
  public  static  void  Main()  {
    char ch;

    for ( ;;) {
      ch  =  (char)  Console.Read();//  Считывание  символа.
      if (ch == 'q' ) break;
    }
    Console.WriteLine("Бы нажали клавишу q!");
  }
}
```

Работа этого бесконечного цикла прекращается с помощью оператора `break`.

Если оператор `break` используется внутри вложенных циклов, то он прекращает работу только внутреннего цикла. Например:

```
//  Использование  оператора  break  во  вложенных  циклах.

using System;
```

```
class  Break3  {
 public  static  void  Main()  {
    int count1 = 0;
      Console.WriteLine("В программе подсчитывается, сколько раз \n"+
                  "был выполнен каждый из циклов.");

    for(int i=0; i<3; i++)  {
      count1++;
      Console.WriteLine("Количество проходов внешнего цикла: " + count1);
      Console.Writie(" Количество проходов внутреннего цикла: ");

      int t = 1;
      while(t < 100) {
        if(t == 11) break; // Оператор прекратит выполнение цикла,
                           // если переменная t примет значение 11.
        Console.Write(t  +  "  ");
        t++;
       }
      Console.WriteLine();
    }
    Console.WriteLine("Внешний цикл завершен.");
  }
}
```

В результате выполнения этой программы на экран будет выведена следующая информация:

```
В программе подсчитывается, сколько раз
был выполнен каждый из циклов.
Количество проходов внешнего цикла: 1
     Количество проходов внутреннего цикла: 1 2 3 4 5 6 7 8 9 10
Количество проходов внешнего цикла: 2
     Количество проходов внутреннего цикла: 1 2 3 4 5 6 7 8 9 10
Количество проходов внешнего цикла: 3
     Количество проходов внутреннего цикла: 1 2 3 4 5 6 7 8 9 10
Внешний цикл завершен.
```

Как видите, использование оператора `break` во внутреннем цикле приводит к остановке только внутреннего цикла. Внешний цикл продолжает работать до полного завершения.

Ответы профессионала

Вопрос. В Java операторы `break` и `continue` могут использоваться с метками. Поддерживается ли такое использование операторов в C#?

Ответ. Нет, разработчики *C#* не наделили операторы `break` и `continue` такими возможностями, эти операторы в C# работают так же, как в языках C и C++. Дело в том. что в языке Java не поддерживается оператор `goto`, поддерживаемый в C#. поэтому в Java операторам `break` п `continue` необходимы дополнительные возможности для компенсации отсутствия оператора `goto`.

Необходимо запомнить еще две особенности использования оператора `break`. Во-первых, в цикле может встречаться больше одного оператора `break`. Будьте внимательны, слишком большое количество операторов `break` и неправильно определенные

условные выражения дли этих операторов могут привести к некорректной работе программы. Во-вторых, оператор `break`, останавливающий работу оператора `switch`, влияет только на этот оператор `switch` и не влияет на работу внешних циклов.

Использование оператора `continue`

В C# существует возможность «заставить» цикл принудительно перейти к следующей итерации (следующему выполнению последовательности операторов, определенных в теле цикла). Это осуществляется с помощью оператора `continue`, который является дополнением к оператору `break`. Например, в следующей программе оператор `continue` используется для вывода на экран четных чисел из диапазона от 0 до 100:

```
   // В программе демонстрируется использование оператора continue.

using System;

class ContDemo {
  public static void Main() {
    int i;

    // Вывод на экран четных чисел,
    // находящихся в диапазоне от 0 до 100.
    for(i = 0; i<=100; i++) {
      if((i%2) !" 0) continue; // Следующая итерация.
      Console.WriteLine(i+ " ");
    }
  }
}
```

Если значение переменной i нечетное, выполняется оператор continue.

В этой программе на экран выводятся только четные числа, поскольку присвоение переменной `i` нечетного значения приведет к выполнению следующей итерации цикла с пропуском вызова метода `WriteLine()`.

В циклах `while` и `do-while` использование оператора `continue` приводит к передаче управления непосредственно условному выражению, после чего выполнение цикла продолжается. В цикле `for` сначала выполняется итерационное выражение цикла, затем оценивается выражение условия, а уже после этого продолжается выполнение цикла.

C# предоставляет возможность большого выбора циклов, которые удовлетворяют требованиям большинства приложений. Если же вы столкнетесь с редкой ситуацией, когда необходимо в определенный момент прекратить выполнение данной итерации Цикла, то сможете сделать это, используя оператор `continue`.

Оператор `goto`

Оператор `goto` в C# является оператором безусловного перехода. Если он встречается в коде, контроль над выполнением программы переходит в то место, которое указано оператором `goto`. Ранее многие программисты избегали использования этого оператора, поскольку в результате его применения программы становились неструктурными и слишком запутанными. Но иногда оператор `goto` может быть очень полезным. Необходимо признать, что в программировании обычно отсутствуют ситуации,

которые требуют обязательного использования этого оператора, скорее, он представляет собой дополнение к инструментарию программиста. В нашей книге оператор goto рассматривается только и этой главе и больше нигде не используется.

Для оператора goto должна быть определена метка, причем ее определение должно быть выполнено в рамках того же метода, в котором она используется оператором goto. Метка представляет собой действительный идентификатор, за которым следует двоеточие. Например, запись цикла, который должен выполнить 100 итераций, при использовании оператора goto и метки выглядит следующим образом:

```
x = 1;
loopl:
   X++;
   if (x < 100) goto loopl;
```

Оператор goto можно эффективно использовать для выхода из глубоко вложенной процедуры. Вот простой пример его использования:

```
/*
  В программе демонстрируется использование оператора goto.
*/

using System;

class Use_goto {
  public static void Main() {
    int i=0, j=0, k=0;

    for (i=0 ; i < 10; i++) {
      for(j=0;j < 10;j++) {
        for(k=0;k < 10;k++) {
          Console.WriteLine("i, j, k: " + i + " " + j + " " + k);
          if(k == 3) goto stop;
        }
      }
    }

stop:
    Console.WriteLine("Завершение выполнения цикла! i, j, k: " + i
                      + ", " + j + " " + k);

  }
}
```

Ниже представлен результат выполнения этой программы.

```
i, j, k: 0 0 0
i, j, k: 0 0 1
i, j, k: 0 0 2
i, j, k: 0 0 3
Завершение выполнения цикла! i, j, k: 0, 0 3
```

Если бы мы исключили из программы оператор goto, то пришлось бы трижды использовать операторы if и break (то есть в данном примере оператор goto упрощает код). Этот пример приведен для того, чтобы вы могли представить себе ситуации, в которых применение оператора goto оправданно.

Минутный практикум

1. Что происходит при выполнении оператора `break` в пределах цикла?
2. Какие действия выполняются оператором `continue`?
3. Какой оператор в C# является оператором безусловного перехода?

Проект 3-3. Завершение создания справочной системы C#

`Help3.cs`

В этом проекте мы завершаем создание справочной системы C#, над которой начали работать в предыдущих проектах. В последней версии добавляется синтаксис операторов `break`, continue и `goto`, а также создается возможность многократного обращения к справочной системе. Осуществляется это посредством добавления внешнего цикла, который выполняется до тех пор, пока пользователь не введет символ `q`.

Пошаговая инструкция

1. Скопируйте код файла `Help2.cs` в новый файл, назовите этот файл `Help3.cs`.
2. Поместите весь код программы внутрь бесконечного цикла `for`. Для прерывания цикла используйте оператор `break`, который должен выполняться при вводе с клавиатуры символа `q`. Поскольку весь код программы помещен внутрь цикла `for`, выход из цикла будет приводить к завершению работы программы.
3. Измените цикл, предназначенный для отображения меню, следующим образом.

```
do {
  Console.WriteLine ("Справка по синтаксису: ");
  Console.WriteLine(" 1. Оператор if");
  Console.WriteLine(" 2. Оператор switch");
  Console.WriteLine(" 3. Цикл for");
  Console.WriteLine(" 4. Цикл while”);
  Console.WriteLine(" 5. Цикл do-while");
  Console.WriteLine(" 6. Оператор break");
  Console.WriteLine(" 7. Оператор continue");
  Console.WriteLine(" 8. Оператор goto\n");
  Console.WriteLine("Введите порядковый номер оператора или цикла: ");
  Console.WriteLine("Для завершения работы программы введите символ q.");
  do {
    choice = (char) Console.Read();
  } while(choice == '\n' | choice == '\r');
} while ( choice < '1'   | choice > '8'   & choice != 'q');
```

---

1. Выполнение оператора `break` в пределах цикла приводит к остановке цикла. Управление программой передается первой строке кода, следующей за циклом.
2. При выполнении оператора `continue` происходит немедленный переход управления к следующей итерации цикла без выполнения оставшегося кода.
3. В C# оператором безусловного перехода является оператор `goto`.

Обратите внимание, что теперь в этот цикл включена возможность выбора синтаксиса операторов `break`, `continue` и `goto`, кроме того, в нем учитывается возможность введения пользователем символа `q`.

4. Расширьте оператор `switch`, чтобы он включал ветви `case`, отображающие синтаксис операторов `break`, continue и `goto`, как это показано ниже.

```
case '6':
  Console.WriteLine("Оператор bteak:\n");
  Console.WriceLine("creak;");
  break;
case '7':
  Console.WriteLine("Оператор continue:\n");
  Console.WriteLine("continue;");
  break;
case '8':
  Console.WriteLine("Оператор goto:\n");
  Console.WriteLine("goto     label;”);
  break;
```

5. Далее представлен полный листинг программы `Help3.cs`.

```
/*
   Проект 3-3

   Заключительная версия справочной системы C#, к которой можно обращаться
   многократно.
*/

using System;

class Help3 {
  public static void Main() {
    char choice;

    for (;;) {
      do {
        Console.WriteLine("Справка по синтаксису: ");
        Console.WriteLine("    1. Оператор if");
        Console.WriteLine("    2. Оператор switch");
        Console.WriteLine("    3. Цикл for");
        Console.WriteLine("    4. Цикл while");
        Console.WriteLine("    5. Цикл do-whiie");
        Console.WriteLine("    6. Оператор break");
        Console.WriteLine("    7. Оператор continue");
        Console.WriteLine("    8. Оператор goto\n");
        Console.WriteLine("Введите порядковый номер оператора или цикла: ");
        Console.WriteLine ("Для завершения работы программы " +
                           "введите символ q.");
        do {
          choice = (char) Console.Read();
        } while (choice == '\n' | choice == '\r');
      } while( choice < '1' | choice > '8' & choice != 'q');

      if (choice == 'q') break;

      Console.WriteLine("\n");
```

```
      switch (choice) {
        case '1':
          Console .WnteLine ( "Оператор i f : \ n” ) ;
          Console . WnteLine ("if (conaition) statemenc; ") ;
          Console.WriteLine("else statement;"I;
          break;
        case ' 2 ' :
          Console -WriteLme ("Оператор switch:\n" ) ;
          Console.WriteLine("switch(expression) (") ;
          Console.WriteLine(" case constant:");
          Console.Wri teLine("       statement sequence”I ;
          Console .WriteLme ( "      break;");
          Console.WriteLine{” // ...”);
          Console.Wri teLine(";") ;
          break;
        case ’3’:
          Console.WriteLine ("Цикл for:\n");
          Console.Write("for(init;condition; iteration) ") ;
          Console.WriteLine(" statement;”);
          break;
        case '4':
          Console.WriteLine(“Цикл while:\n");
          Console -Wri teLine("while(condition) statement;") ;
          break;
        case ’5’:
          Console . WriteLme ( "Цикл do-whiic : \n" ) ;
          Console-WnteLine ("do {");
          Console.Wri teLine(“ statement;");
          Console . WnteLine (" > while (condition);");
          break;
        case '6’ :
          Console.WriteLine("Оператор break:\n");
          Console.WriteLine("break;");
          break;
        case '7':
          Console.WriteLine("Оператор continue:\n");
          Console.WriteLine("continue;”) ;
          oreak;
        case ' 8 ' :
          Console.WriteLine("Оператор goto:\n");
          Console.WriteLine("goto laoel;");
          break;
      }
      Console.WriteLine();
    }
  }
}
```

Ниже показан один **НЈ ВОЗМОЖНЫХ** результатов выполнения программы.

```
Справка по синтаксису:
  1. Оператор if
  2. Оператор swiLch
  3. Цикл for
  4. Цикл while
  5.  Цикл  do-while
  6. Оператор break
  7.  Оператор continue
```

```
   8. Оператор goto
Введите порядковый номер оператора или цикла:
Для завершения работы программы введите символ q: 1
Оператор if:

it(condition)   statement;
else statement;

Справка по синтаксису:
   1. Оператор if
   2. Оператор switch
   3. Цикл for
   4. Цикл while
   5. Цикл do-while
   6. Оператор break
   7. Оператор continue
   8. Оператор goto
Введите порядковый номер оператора или цикла:
Для завершения работы программы введите символ q: 6

Оператор break:

break;

Справка по синтаксису:
   1. Оператор if
   2. Оператор switch
   3. Цикл for
   4. Цикл while
   5. Цикл do—while
   6. Оператор break
   7. Оператор continue
   8. Оператор goto
Введите порядковый номер оператора или цикла:
Для завершения работы программы введите символ q: q
```

Вложенные циклы

Как видно из предыдущих примеров, один никл можно вложить внутрь другого. Вложенные никлы используются для решения разнообразных задач и являются мощным инструментом в программировании. Поэтому, прежде чем завершить описание операторов C#, рассмотрим еще один пример вложенного цикла. В следующей программе используется вложенный цикл `for` для нахождения делителей чисел, находящихся в диапазоне от 2 до 100:

```
/*
   В программе демонстрируется использование вложенного цикла для нахождения
   делителей целых чисел, находящихся в диапазоне от 2 до 100.
*/

using System;

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

```
      for (int i=2;i <= 100;i++) {
       Console.Write("Делители числа " + i + ": ");
       for(int j - 2;j < i; j++)
          if((i%j) == 0) Console.Write(j + " ");
       Console.WriteLine();
      }
   }
}
```

Ниже представлена часть результатов выполнения программы.

```
Делители числа 2:
делители числа 3:
Делители числа 4: 2
Делители числа 5:
Делители числа 6: 2 3
Делители числа 7:
делители числа 8: 2 4
делители числа 9: 3
Делители числа 10: 2 5
Делители числа 11:
Делители числа 12: 2 3 4 6
Делители числа 13:
Делители числа 14: 2 7
Делители числа 15: 3 5
Делители числа 16: 2 4 8
Делители числа 17:
Делители числа 18: 2 3 6 9
Делители числа 19:
Делители числа 20: 2 4 5 10
```

В этой программе при выполнении внешнего цикла значение переменной `i` увеличивается от 2 до 100. Внутренний цикл успешно проверяет все значения от 2 до значения, которое в данный момент имеет переменная `i`, выводя на экран те из них, на которые значение переменной `i` делится без остатка.

Контрольные вопросы

1. Напишите программу, которая считывает символы, вводимые с клавиатуры, до тех пор, пока не будет введен символ точки (.). Программа должна сосчитать количество введенных пробелов и вывести это число в конце программы.
2. Может ли последовательность кода одной ветви `case` передавать управление программой другой ветви `case` в операторе `switch`?
3. Напишите синтаксис цепочки операторов `if-else-if`.
4. С каким оператором `if` ассоциирован последний оператор `else` в коде

```
lf(x < 10)
    if(y > 100) {
        if (!done) x = z;
        else y = z;
    }
else Console.WriteLine("Ошибка!");//К какому оператору if
                                        // принадлежит его часть else?
```

5. Запишите пример цикла `for`, в итерационном выражении которого число 1000 должно уменьшаться до 0 с шагом -2.
6. Является ли действительным следующий фрагмент кода:

```
for(int i - 0; i < num; i++)
    sum += i;

count = i;
```

7. Какие действия выполняет оператор `break`?
8. Рассмотрите следующий фрагмент кода. Какая информация будет выведена на экран после выполнения оператора `break`?

```
for(i = 0; i < 10; i++) {
    while(running)   {
        if(x<y) break;
        // ...
    }
    Console.WriteLine("Цикл while завершил свою работу.");
}
 Console.Writeline("Цикл for завершил свою работу.");
```

9. Как будет оформлен вывод на экран значения переменной 1?

```
for(int 1 = 0;i < 10;i++) {
    Console.Write(i + "  ");
    if((i%2)  == 0) continue;
    Console.WriteLine();}
```

10. Итерационное выражение в цикле `for` не всегда должно изменять контрольную переменную цикла на фиксированную величину. Контрольная переменная цикла может изменяться произвольно. Напишите программу, в которой цикл `for` используется для вывода на экран геометрической прогрессии 1, 2, 4, 8, 16, 32, 64 и так далее.

11. Числовые значения символов нижнего регистра в коде ASCII отличаются от значении символов верхнего регистра на величину 32. Следовательно, для конвертирования символа нижнего регистра и символ верхнего регистра необходимо вычесть из его значения число 32. Используя эту информацию, напишите программу, которая читает символы, вводимые с клавиатуры. Программа должна конвертировать все символы нижнего регистра в символы верхнего регистра и наоборот, выводя на экран результат. При этом все остальные символы остаются неизменными. Программа должна прекращать работу, когда пользователь вводит символ точки (.). В завершение работы программа должна вывести информацию о количестве измененных символов.

Классы, объекты и методы

- Определение класса
- Создание объектов
- Создание методов
- Передача методу параметров
- Возвращение методом значений
- Использование конструкторов
- Оператор `new`
- Деструкторы и «сборка мусора»
- Ключевое слово this

Прежде чем продолжать изучение языка C#, необходимо рассмотреть основную структуру объектно-ориентированного программирования — класс. Класс является фундаментом, на котором построен весь язык C#. В классе определены данные и код, работающий с этими данными. Это очень обширная и важная тема, поэтому внимательно изучите данную главу. Только поняв, что представляют собой классы, объекты и методы, вы сможете писать более сложные и совершенные программы на C#.

Основные понятия класса

Весь активный процесс C#-программы происходит в пределах класса, поэтому с самого начала в программах этой книги использовались классы. Понятно, что они были самыми простыми и вы смогли познакомиться только с малой частью их возможностей. Классы, рассматриваемые дальше, обладают гораздо большей мощью. В этом раздое мы подробно остановимся на основных понятиях класса.

Класс является основой, для создания объектов. В классе определяются данные и код, который работает с этими данными. Объекты являются *экземплярами* класса. Непосредственно инициализация переменных в объекте *(переменных экземпляра)* происходит в конструкторе. В классе могут быть определены несколько конструкторов, то есть класс является набором проектов, которые определяют, как строить объект. Очень важно понимать разницу между классом и объектом: класс является логической абстракцией до тех пор, пока не будет создан объект и не появится физическая реализация этого класса в памяти компьютера.

Методы и переменные, составляющие класс, называются *членами* класса.

Общий синтаксис класса

При определении класса объявляются данные, которые он содержит, и код, работающий с этими данными. Самые простые классы могут содержать только код или только данные, но в реальных программах классы включают обе эти составляющие.

Данные содержатся в переменных экземпляра, которые определены классом, а код содержится в методах. Важно с самого начала отмстить, что в C# определены несколько специфических разновидностей членов класса. Это — переменные экземпляра, статические переменные, константы, методы, конструкторы, деструкторы, индексаторы, события, операторы и свойства. На данном этапе мы ограничимся описанием переменных экземпляра и методов, являющихся основными элементами класса. Позже в этой главе будут рассмотрены конструкторы и деструкторы. О других типах членов класса мы расскажем в следующих главах этой книги.

При создании (определении) класса вначале указывается ключевое слово `class`. Ниже представлен общий синтаксис определения класса, содержащий только переменные экземпляра и методы.

```
class classname {
  //  объявление  переменных  экземпляра
  access  type  var1;
  access type var2;
  // ...
  access type varN;
  // объявление методов
  access ret-type methodl(parameters){
```

```
        // тело метода
    }
    access ret-type method2(parameters)1
        // тело метода
    }
    /// ...
    access ret-type methodK(parameters){
        // тело метода
   }
}
```

Обратите внимание, что объявление каждой переменной и метода начинается с подстановочного слова `access`, вместо которого объявляется модификатор, указывающий, как осуществляется доступ к данному члену класса. Как уже говорилось в главе 1, использование модификатора `private` указывает на то, что члены класса доступны только для членов этого класса, а использование модификатора `public` — на то, что они открыты для кода, определенного за пределами данного класса. При определении члена класса модификатор указывать не обязательно, по умолчанию его отсутствие означает, что этот член класса имеет модификатор `private`. Члены класса, объявленные как `private`, могут использоваться только другими членами этого класса. В программах данной главы все члены класса определены как имеющие тип доступа `public`. Это означает, что они могут быть использованы любым другим кодом, даже определенным за пределами класса. Подробнее мы рассмотрим модификаторы в следующих главах.

Не существует строгого синтаксического правила, ограничивающего сферу действий, выполняемых классом, или разнообразие сохраняемых данных, но в хорошо написанной программе класс определяет одну и только одну логическую сущность. Например, класс, в котором хранятся имена пользователей и их телефонные номера, как правило, не должен хранить данные фондовой биржи, информацию о среднемесячных температурах в Антарктиде, расписание авиарейсов либо другую логически не связанную с телефонным справочником информацию.

До сих пор используемые нами классы имели только один метод — `Main()`. В этой главе мы рассмотрим, как создавать другие методы. Необходимо отметить, что определение метода `Main()` не является неотъемлемой частью синтаксиса класса, этот метод необходим только в классе, с которого должна начинаться программа.

Определение класса

Любую тему легче всего изучать, используя практические примеры. Мы разработаем класс, который инкапсулирует информацию о транспортных средствах, таких как легковые автомобили, фургоны и грузовики, и назовем его `Vehicle`. Этот класс будет хранить данные о транспортных средствах — количество пассажиров, которое способен перевозить автомобиль, объем топливного бака и расстояние (в милях), которое этот автомобиль может проехать, используя один галлон топлива.

Первая версия класса `vehicle`, в котором определены три переменные экземпляра `passengers`, `fuelcap` и `mpg`, представлена ниже. Обратите внимание, что класс `Vehicle` не содержит никаких методов, то есть эта версия класса содержит только данные. (Далее в следующих версиях будут добавляться и методы.)

```
class Vehicle {

  public int   passengers;  // Количество пассажиров.
  public int   fuelcap;     // Емкость топливного бака (в галлонах).
  public int   mpg;         // Расстояние (в милях), которое
                            // автомобиль может проехать, используя один
                            // галлон топлива.
}
```

Приведенное выше определение переменных экземпляра можно рассматривать как общий способ объявления переменных. Синтаксис объявления переменных экземпляра таков:

```
access type var-name;
```

Здесь подстановочное слово `access` — это модификатор, слово `type` — тип переменной, а словосочетание `var-name` — имя переменной. Таким образом, синтаксис объявления переменной экземпляра аналогичен синтаксису объявления локальной переменной, за исключением модификатора. В классе `Vehicle` объявление переменных начинается с модификатора `public`, это позволяет переменным быть доступными для кода, определенного вне данного класса.

С помощью ключевого слова `class` создается новый тип данных. В нашем случае новый тип данных называется `Vehicle`. Это имя используется для объявления объектов типа `Vehicle`. Помните, что объявление класса с помощью ключевого слова `class` даст только описание типа, но физический объект при этом не создается. Поэтому выполнение приведенного выше кода не приведет к возникновению объектов типа `Vehicle`.

Для того чтобы создать объект типа `Vehicle`, необходимо использовать оператор `new`, о котором мы расскажем далее в этой главе. Например,

```
Vehicle minivan = new Vehicle(); // Создание объекта типа Vehicle,
                                 // который называется minivan.
```

После выполнения этого оператора будет создан экземпляр класса `Vehicle` — объект `minivan`.

При создании экземпляра класса создается объект, который содержит собственную копию каждой переменной экземпляра, определенной классом. Следовательно, каждый объект класса `Vehicle` будет содержать свои собственные копии переменных экземпляра `passengers`, `fuelcap` и `mpg`. Для доступа к этим переменным используемся оператор *точка* (`.`). Этот оператор связывает имя объекта с именем члена класса. Приведем общий синтаксис данного оператора:

```
object.member
```

Как видите, имя объекта указывается слева от точки, а имя члена класса — справа. Например, для присваивания переменной `fuelcap` объекта `minivan` значения 16 используется следующий оператор:

```
minivan.fuelcap = 16;
```

Оператор точка (`.`) может использоваться для получения доступа как к переменным экземпляра, так и к методам.

Ниже представлен весь код программы, в которой используется класс `Vehicle`.

```
/* В программе демонстрируется использование класса Vehicle.

   Назовите этот файл UseVehicle.cs
```

```
*/
using System;

class Vehicle {
   public int    passengers;  //  Количество пассажиров.
   public int    fuelcap;     //  Емкость топливного бака (в галлонах).
   public int    mpg;         //  Расстояние (в милях), которое данный автомобиль
                               // может проехать, используя один галлон топлива.}

// В этом классе создается объект класса Vehicle,
class VehicleDemo {
  public static void Main() {
    Vehicle minivan = new Vehicle();
    int range;

    // Переменным объекта minivan присваиваются значения,
    minivan.passengers = 7;
    minivan.fuelcap = 16;
    minivan.mpg = 21;

    // Вычисление максимального расстояния, которое может проехать
    // данный автомобиль, имея полный топливный бак.
    range = minivan.fuelcap * minivan.mpg;

    Console.WriteLine("Миниавтобус может перевезти " +
                      minivan.passengers + " пассажиров на расстояние " +
                      range + " миль.");
  }
}
```

Эта строка кода создаст экземпляр класса Vehicle, который называется minivan.

Обратите внимание на использование оператора точка (.) для доступа к члену класса.

В этой программе содержатся два класса `Vehicle` и `VehicleDemo`. Внутри класса `VehicleDemo` в методе `Main()` создается экземпляр класса `Vehicle`, называемый `minivan`. Затем код, определенный и методе `Main()`, получает доступ к переменным экземпляра (объекта `minivan`), присваивая им значения и используя эти значения в вычислениях. Важно понимать, что классы `Vehicle` и `VehicleDemo` являются различными, самостоятельными классами. Единственная связь между ними состоит в том, что в одном классе создается экземпляр другого класса. Хотя эти классы являются самостоятельными, код внутри класса `VehicleDemo` может иметь доступ к членам класса `vehicle`, поскольку переменные класса `Vehicle` объявлены как `public`. Без указания этого модификатора доступ к переменным `passengers`, `fuelcap` и `mpg` мог бы осуществляться только кодом, определенным в пределах класса `Vehicle`, а класс `VehicleDemo` не имел бы возможности их использовать.

Рассматриваемая программа была помещена в файл `Usevehicle.cs`, поэтому при ее компилировании будет создан файл `Usevehicle.exe`. Оба класса, `Vehicle` и `VehicleDemo`, автоматически станут частью исполняемого файла. Результат выполнения этой программы будет следующим:

```
Миниавтобус может перевезти 7 пассажиров на расстояние 336 миль.
```

Классы `Vehicle` и `VehicleDemo` не обязательно должны находиться в одном и том же исходном файле. Можно поместить каждый класс в отдельный файл, назвав их, например. `Vehicle.cs` и `VehicleDemo.cs`. В этом случае нужно задать в командной строке команду

```
csc Vehicle.cs VehicleDemo.cs
```

при выполнении которой оба файла (оба класса, содержащиеся и них) будут не только скомпилированы, но и объединены.

При работе в интегрированной среде Visual C++IDE необходимо добавить к программе оба эти файла, а затем их построить.

Прежде чем перейти к изучению следующего материала, еще раз вернемся к основному принципу создания объектов — каждый объект имеет собственные копии переменных экземпляра, определенных их классом. Следовательно, значения переменных одного объекта могут отличаться от значений переменных другого объекта. Например, если существуют два объекта `Vehicle`, то каждый из них имеет свои собственные копии переменных `passengers`, `fuelcap` и `mpg`, значения которых могут различаться. Следующая программа демонстрирует использование этого принципа.

```
   // В этой программе создаются два объекта класса Vehicle.

using System;

class Vehicle {
  public int passengers; // Количество пассажиров.
  public int fuelcap;    // Емкость топливного бака (в галлонах).
  public int mpg;        // Расстояние (в милях), которое данный автомобиль
                            // может проехать, используя один галлон топлива.


   // В этом классе создаются два объекта класса Vehicle,
class TwoVehicles {
  public static void Main() {
    Vehicle minivan = new Vehicle();
    Vehicle sportscar = new Vehicle();

    int range1, range2;

    // Переменным объекта minivan присваиваются значения.
    minivan.passengers = 7;
    minivan.fuelcap = 16;
    minivan.mpg = 21;

    // Переменным объекта sportscar присваиваются значения.
    sportscar.passengers = 2;
    sportscar.fuelcap = 14;
    sportscar.mpg = 12;

    // Вычисление максимального расстояния, которое может проехать
    // каждый из автомобилей, имея полный топливный бак.
    range1 = minivan.fuelcap * minivan.mpg;
    range2 = sportscar.fuelcap * sportscar.mpg;

    Console.WriteLine("Микроавтобус может перевезти " +
                      minivan.passengers + " пассажиров на расстояние " +
                      range1 + " миль.");

    Console.WriteLine("Спортивный автомобиль может перевезти " +
                      sportscar.passengers + " пассажира на расстояние \n" +
                      range2 + " миль.");
    }
  }
```

Помните, что переменные `minivan` и `sportcar` ссылаются на различные объекты.

Ниже приведен результат выполнения этой программы.

```
Миниавтобус может перевезти 7 пассажиров на расстояние 336 миль.
Спортивный автомобиль может перевезти 2 пассажиров на расстояние 168 миль.
```

Данные объекта `minivan` полностью независимы отданных, содержащихся в объекте `sportcar`. На рис. 4.1 представлена структура этих объектов.

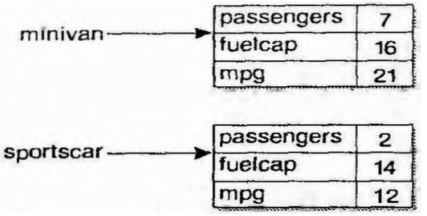


Рис. 4.1. Переменные экземпляра одного объекта независимы от переменных экземпляра другого объекта

Минутный практикум

1. Из каких компонентов состоит класс?
2. Какой оператор используется для получения доступа к членам класса?
3. Собственные копии чего имеет каждый объект?

Как создаются объекты

В предыдущей программе для объявления объекта типа `Vehicle` была использована следующая строка кода:

```
Vehicle minivan = new Vehicle();
```

В ней выполняются два действия. Во-первых, объявляется переменная типа `Vehicle`, называемая `minivan`. Эта переменная не определяет объект (то есть не является непосредственно объектом), она только *ссылается* на объект. Во-вторых, при выполнении оператора `new` создается физический объект, а переменной `minivan` присваивается ссылка на этой объект. Следовательно, после выполнения данной строки кода переменная `minivan` будет ссылаться на объект типа `Vehicle`.

Оператор `new` *динамически* (во время работы программы) *выделяет* память для объекта и возвращает ссылку на эту область памяти. То есть ссылка является адресом памяти, выделенной объекту оператором `new`. В дальнейшем эта ссылка хранится в переменной. Следовательно, всем объектам в C# память должна выделяться динамически.

Два шага, совмещенные в предыдущем операторе (строке кода), для большей наглядности могут быть переписаны в двух операторах:

```
Vehicle minivan;              // Объявление ссылки на объект.
ir.inivan = new Vehicle();    // Выделение памяти для объекта Vehicle и возвращение
                              // ссылки на этот объект переменной minivan.
```

1. Класс состоит из кода и данных.
2. Для доступа к членам класса используется оператор точка (.).
3. Каждый объект имеет собственные копии переменных экземпляра.

В первой строке кода переменная `minivan` объявляется как ссылка на объект типа `Vehicle`. То есть переменная `minivan` может ссылаться на объект, но не является самим объектом. На этом этапе данная переменная хранит значение `null`, означающее, что связь между ней и объектом отсутствует. В следующей строке кода создается новый объект класса `Vehicle`, а переменной `minivan` присваивается ссылка на этот объект, теперь переменная `minivan` связана с объектом.

Поскольку доступ к объектам класса осуществляется с помощью ссылки, классы иначе называют *ссылочными типами.* Ключевое отличие между обычными и ссылочными типами состоит в значениях, хранимых переменными каждого типа. Переменная обычного типа сама хранит значение. Например, после выполнения операторов

```
int x;
x = 10;
```

переменная `x` хранит значение 10, поскольку `x` является переменной типа `int`, то есть переменной обычного типа. Но после выполнения оператора

```
Vehicle minivan = new Vehicle();
```

переменная `minivan` самостоятельно не хранит объект, а хранит только ссылку на него.

Переменные ссылочного типа и оператор присваивания

При выполнении операции присваивания переменные ссылочного типа действуют иначе, чем переменные обычного типа (например, типа `int`). Когда значение одной переменной обычного типа вы присваиваете другой переменной обычного типа, то ситуация проста. Переменная, находящаяся слева от оператора присваивания, получает *копию значения* переменной, указанной справа. Когда же значение одной переменной ссылочного типа вы присваиваете другой переменной, ситуация гораздо сложнее, поскольку вы присваиваете переменной ссылку на другой объект. Например, рассмотрим следующий фрагмент кода:

```
vehicle car1 = new Vehicle();
Vehicle car2 = car1;
```

Может показаться, что переменные `car1` и `car2` относятся к различным объектам, но это не так. На самом деле обе переменные `car1` и `car2` ссылаются на *один и тот же* объект. При присваивании переменной `car2` значения переменной `car1`, переменной `car2` присваивается ссылка на тот же объект, на который ссылается переменная `car1`. Следовательно, доступ к объекту можно осуществлять как с помощью переменной `car1`, так и с помощью переменной `car2`. Например, после выполнения операции присваивания

```
car1.mpg = 26;
```

оба оператора `WriteLine()` выведут одинаковое значение — 26.

```
Console.WriteLine(car1.mpg);
Console.WriteLine(car2.mpg);
```

Переменные `car1` и `car2` не связаны каким-либо способом, хотя и ссылаются на один объект. Например, после выполнения следующего фрагмента кода переменная `car2` получает ссылку на объект, на который ссылается переменная `car3`.

```
Vehicle car1 = new Vehicle();
Vehicle car2 = car1;
Vehicle car3 = new Vehicle();
car2 = car3;   // Теперь переменные car2 и car3 ссылаются
               // на один и тот же объект.
```

Объект, на который ссылается переменная `car1`, остается без изменений.

Минутный практикум

1. Что происходит, когда значение переменной ссылочного типа присваивается другой переменной ссылочного типа?
2. Предположим, что создан класс с именем `MyClass`. Покажите, как создать объект с именем `ob`.
3. Чем отличаются переменные ссылочного типа от переменных обычного типа?

Методы

Как уже говорилось, переменные экземпляра и методы являются двумя основными составляющими классов. До этого момента в нашей программе класс `Vehicle` содержал только данные и не содержал методов. Хотя классы, содержащие только данные, являются действительными, большинство классов содержит и методы. Методы — это подпрограммы, которые управляют данными, определенными в классе, и во многих случаях обеспечивают доступ к данным. Обычно остальные части программы взаимодействуют с классом при помощи его методов.

Метод содержит один и более операторов. В профессионально написанном C#-коде каждый метод выполняет только одну задачу. В качестве имени метода может использоваться любой действительный идентификатор. Ключевые слова не могут быть именами методов, имя `Main()` является зарезервированным для метода, с которого начинается выполнение программы.

В тексте программы методы обозначаются следующим образом: после имени метода следует пара круглых скобок. Например, если имя метода `getvar`, то при его вызове и программе он будет написан как `getvar()`. Такая форма написания применяется дли того, чтобы в программах можно было различать имена переменных и методов.

Общи и синтаксис метода выглядит так:

```
access ret-type name (parameter ter-list) {
    // тело метода
}
```

Здесь подстановочное слово `access` — это модификатор, который указывает, какие части вашей программы могут иметь доступ к методу. Как уже говорилось, модификатор не является обязательным, если он отсутствует, метод доступен только в

---

1. Когда значение переменной ссылочного типа присваивается другой переменной ссылочного типа, обе переменные будут ссылаться на один и тот же объект. При этом копирования объекта не происходит.

2. `MyClass ob = new MyClass();`

3. Переменная обычного типа непосредственно хранит присвоенное ей значение, переменная ссылочного типа хранит только ссылку на объект.

пределах класса, в котором он объявлен. Пока мы будем объявлять все методы как `public`, поэтому они могут быть вызваны кодом из любого места программы. Словосочетание `ret-type` в синтаксисе указывает тип данных, возвращаемых методом. Эти могут быть как данные любого действительного типа, так и объекты любого, созданного вами класса. Если метод не возвращает никакого значения, он должен быть указан как имеющий тип `void`. Вместо слова `name` указывается имя метода. Это может быть любой действительный идентификатор, не повторяющий имена, которыми были названы другие члены класса в той же области видимости. После имени метода следует словосочетание `parameter-list` (список параметров) — последовательность разделенных запятыми пар «тип-идентификатор». То есть с помощью параметров методу передаются из программы значения, тип которых обязательно должен быть указан. Параметрами в основном являются переменные, которые принимают значения *аргументов*, передаваемых методу при ею вызове. Если метод не имеет параметров, то список параметров будет пустым.

Добавление метода к классу `Vehicle`

Мы уже говорили ранее, что методы класса обычно работают с данными и обеспечивают доступ к данным класса. Теперь вспомним, что в методе `Main()` из предыдущею примера для вычисления расстояния, которое может проехать автомобиль с полным топливным баком, значение расстояния, которое автомобиль может проехать, используя один галлон топлива, умножалось на значение емкости топливного бака. Хотя технически мы все выполнили правильно, этот способ не является оптимальным при выполнении такого вычисления. Максимальную дальность поездки лучше всего вычислять в самом классе `Vehicle`, поскольку максимальное расстояние, которое может проехать автомобиль, зависит от емкости его топливного бака и расхода топлива, а обе эти величины инкапсулированы в классе `Vehicle`. Кроме того, при добавлении к классу `Vehicle` метода, и котором вычисляется дальность поездки, улучшается объектно-ориентированная структура класса.

Для того чтобы добавить метод к классу `Vehicle`, его нужно определить при создании класса. Например, следующая версия класса `Vehicle` содержит метод `range()`, выводящий данные о расстоянии, пройденном автомобилем:

```
// В этой программе в классе Vehicle определен метод range.

using System;

class Vehicle {
  public int passengers;   // Количество пассажиров.
  public int fuelcap;      // Емкость топливного бака (в галлонах).
  public int mpg;          // Расстояние (в милях), которое данный
                           //  автомобиль  может  проехать,  используя  один
                           // галлон топлива.

  // Определение метода range.
  public void range() {
    Console.WriteLine("Этот автомобиль может проехать с полным "+
                      " топливным баком "+ fuelcap * mpg + " миль.");
  }
}

class AddMeth {
```

Метод `range()` принадлежит классу `Vehicle`.

Обратите внимание, что переменные `fuelcap` и `mpg` используются непосредственно без использования оператора точка (.).

```
  public static void Main() {
    Vehicle minivan = new Vehicle();
    Vehicle sportscar = new Vehicle();

    // Значения присваиваются переменным объекта minivan.
    minivan.passengers = 7;
    minivan.fuelcap = 16;
    minivan.mpg = 21;

    // Значения присваиваются переменным объекта sportscar.
    sportscar.passengers = 2;
    sportscar-fuelcap = 14;
    sportscar.mpg = 12;


    Console.Write("Микроавтобус может перевозить " + minivan.passengers +
                  " пассажиров.\n");

    minivan.range();    // Вывод на экран значения максимального расстояния,
                        // которое может проехать микроавтобус.

    Console.Write("Спортивный автомобиль может перевозить " +
                  sportscar.passengers + " пассажиров.\n");

    sportscar.range();    // Вывод на экран значения максимального расстояния,
                          // которое может проехать спортивный автомобиль.
  }
}
```

Результат работы этой программы будет следующим: ",

```
Микроавтобус может перевозить 7 пассажиров.
Этот автомобиль может проехать с полным топливным баком 336 миль.
Спортивный автомобиль может перевозить 2 пассажиров.
Этот автомобиль может проехать с полным топливным баком 168 миль.
```

Теперь рассмотрим все ключевые элементы программы. Начнем с метода `range()`. Вот начало объявления метода `range()`:

```
public void range()  {
```

В этой строке кода объявляется метод с именем `range`, не имеющий параметров. Он определен как `public`, что позволяет вызывать его в любой другой части программы, `void` — тип возвращаемого им значения, то есть метод `range()` не возвращает в программу никакого значения. Строка заканчивается открывающей фигурной скобкой, за которой определяются операторы метода.

Тело метода `range()` содержит единственную строку

```
Console.WriteLine("Этот автомобиль может проехать с полным "+
                  " топливным баком "+ fuelcap * mpg + " миль.");
```

Этот оператор выводит значение максимальной дальности поездки, вычисляя его путем умножения значений переменных `fuelcap` и `mpg`. Поскольку каждый объект типа `Vehicle` имеет собственные копии переменных `fuelcap` и `mpg`, при вызове метода `range()` для вычисления дальности поездки используются копии переменных вызывающего объекта.

Метод `range()` заканчивается закрывающей фигурной скобкой. При этом управление передается в точку программы, из которой метод был вызван.

Обратите внимание на оператор

```
minivan.range();
```

внутри метода `Main()`. Этот оператор вызывает метод `range()` для объекта `minivan`, после чего управление передается этому методу. То есть вызывается метод `range()`, который будет работать с переменными объекта `minivan`. Для этого указывается имя объекта, за которым следует оператор точка (`.`). Когда метод выполнит свою работу, управление будет передано в точку программы, из которой метод был вызван. Выполнение программы продолжается со строки кода, следующей за вызовом метода.

В этом случае вызов метода `minivan.range()` выводит информацию о максимальном расстоянии, которое может проехать автомобиль `minivan`, а вызов метода `sportscar.range()` выводит информацию о максимальном расстоянии, которое может проехать автомобиль `sportscar`. Каждый раз при вызове метод `range()` работает с переменными объекта, указанного слева от оператора точка (`.`).

Обратите внимание на очень важную особенность определения метода `range()`: переменные экземпляра `fuelcap` и `mpg` указываются непосредственно (для идентификации переменной не используется оператор точка (`.`) и не указывается имя объекта). Когда метод работает с переменной экземпляра, которая определена в том же классе, что и сам метод, он обращается к переменной без явного указания объекта и без использования оператора точка (то есть в пределах метода нет необходимости указывать имя объекта второй раз). Это означает, что используемые переменные `fuelcap` и `mpg` неявно относятся к копиям переменных, которые определены в объекте, вызывающем метод `range()`.

Возврат управления из метода

Управление возвращается из метода в двух случаях. Это происходит, во-первых, когда встречается закрывающая фигурная скобка этого метода, во-вторых, когда выполняется оператор `return`. Существует две формы оператора `return`, одна форма используется в методах, имеющих тип `void` (которые не возвращают значения), а вторая — в методах, возвращающих значения.

В методах, имеющих тип `void`, выполнение оператора `return` приводит к немедленному завершению работы метода. Поэтому обычно этот оператор указывается после условного выражения, поскольку при определенных условиях выполнение оставшихся операторов метода может не иметь смысла. Оператор return записывается следующим образом:

```
return;
```

Когда он выполняется, управление передается в точку программы, откуда метод был вызван, при этом весь оставшийся в методе код пропускается. В качестве примера рассмотрим следующий метод:

```
public void myMeth() {
  int i;

  for(i=0; i<10; i++)    {
    if(i == 5) return; // Метод прекращает свою работу, когда переменная i
                       // принимает значение 5.
    Console.WriteLine();
  }
}
```

Здесь цикл for выполнится только 5 раз, потому что, когда значение переменной i станет ранным 5, будет выполнен оператор return.

В рамках одного метода допускается наличие более одного оператора return. Чаше всего такие операторы используются к тех методах, которые имеют несколько условных выражении. Например.

```
public void myMeth() {
    // ...
    if(condition_1) return;
    //
    if(condicion_2) return;
    statements;
}
```

Возврат управления из этого метода осуществляется в трех случаях: после выполнения заключительных операторов метода, после выполнения первого условного выражения и после выполнения второго условного выражения. Но если метод имеет слишком много точек выхода, это деструктурирует код. поэтому следует избегать чрезмерного использования операторов return.

Возвращение методом значений

Хотя методы, имеющие тип void, встречаются достаточно часто, большинство методов в C# возвращают значения. Фактически способность возвращать значение является одним из наиболее важных свойств метода. Вы уже встречали пример возвращения методом значения, когда мы использовали функцию Math.Sqrt() для получения значения квадратного корня числа.

Возвращение значений в программировании используется для различных целей. В одних случаях, например, при использовании функции Math.Sqrt(), возвращаемое значение является результатом некоторого вычисления. В других случаях возвращаемое значение может указывать на успех или неудачу операции (возвращается булево значение). Иногда возвращаемое значение может содержать код состояния (возвращается заранее определенное число — -1. () и так далее). Использование возвращаемых методом значений — неотъемлемая часть программирования.

Методы возвращают значение вызывающей подпрограмме, используя следующую форму оператора retutn:

```
return value;
```

Здесь подстановочное слово value — возвращаемое значение.

Способность методов возвращать значения можно использовать для усовершенствования уже знакомого нам метола range(). Вместо вывода информации о максимальном расстоянии, которое может проехать данный автомобиль, новый вариант метода range() будет вычислять значение максимального расстояния и возвращать его программе. Одним из преимуществ такого подхода является возможность использования возвращаемого значения для других вычислений. В следующем варианте программы модифицированный метод range() не выводит на экран соответствующую информацию о данном автомобиле, а возвращает вычисленное значение.

```
// В программе демонстрируется использование метода,
// возвращающего вычисленное значение.
```

```
using System;

class Vehicle {
  public int passengers;     // Количество пассажиров.
  public int fuelcap;        // Емкость топливного бака (в галлонах).
  public int mpg;            // Расстояние (в милях), которое данный
                             // автомобиль может проехать, используя один
                             // баллон топлива.

  // Определение метода, возвращающего значение типа int.
  public int range()   {
    return mpg * fuelcap;
  }
}
class RetMeth {
  public static void Main()      {
    Vehicle minivan = new Vehicle();
    Vehicle sportscar — new Vehicle();

    int range1, range2;

    // Значения присваиваются переменным объекта minivan.
    minivan.passengers = 7;
    minivan.fuelcap = 16;
    minivan.mpg = 21;

    // Значения присваиваются переменным объекта sportscar.
    sportscar.passengers = 2;
    sportscar.fuelcap = 14;
    sportscar.mpg = 12;

    // Вычисление максимального расстояния, которое может проехать
    // данный автомобиль.
    rangel = minivan.range();
    range2 = sportscar.range();


    Console.WriteLine("Микроавтобус может перевезти " + minivan.passengers +
                      " пассажиров на расстояние " + rangel + " миль.");

    Console.WriteLine("Спортивный автомобиль может перевезти " +
      sportscar. passengers + " пассажиров на расстояние " + range2 + " миль.");
  }
}
```

Теперь метл range () нозирашаст значение.

Переменным range1 и range2 присваиваются возвращаемые значения.

Результат выполнения этой программы следующий:

```
Микроавтобус может перевезти 7 пассажиров на расстояние 336 миль.
Спортивный автомобиль может перевезти 2 пассажиров на расстояние 168 миль.
```

В этой программе вызов метода `range()` помещен справа от оператора присваивании. Слева от него находится переменная, которой будет присвоено значение, возвращаемое методом `range()`. Следовательно, после выполнения строки кода

```
rangel = minivam.range();
```

переменной `range1` присваивается знамение, для вычисления которого использовались переменные объекта `minivan`.

Теперь метод range() возвращает значение типа int. Очень важно правильно указать тип возвращаемого методом значения (если возникнет необходимость возврата методом данных типа double, при объявлении метода следует указать именно этот тип возвращаемого значения).

Хотя синтаксис представленной выше программы не содержит ошибок, она написана недостаточно эффективно. В частности, отсутствует необходимость в переменных range1 и range2, поскольку метод range() может быть вызван непосредственно в операторе WriteLine();, как это показано ниже.

```
Console.WriteLine("микроавтобус может перевезти " + mimvan.passengers + "
пассажиров на расстояние " + minivan.range() + " миль.");
```

В этом случае при выполнении оператора WriteLine(); автоматически вызывается метод minivan.range(), а его значение передается методу WriteLine(). Кроме того, вы можете вызвать метод range() в любом месте программы, где необходимо использование значения максимального расстояния данного объекта класса Vehicle. Например, в следующей строке кода производится сравнение значений, возвращаемых методом range().

```
if (v1.range() > v2.range())
Console.WriteLine("Автомобиль v1 с полным топливным баком может проехать"+
                  "\n большее расстояние, чем автомобиль v2.");
```

Ответы профессионала

Вопрос. Я слышал, что компилятор C# может обнаружить код, который не будет выполнен ни при каких условиях. Что это означает?

Ответ. Это действительно так, если компилятор обнаружит в созданном вами методе код, который не может быть выполнен, он выдаст предупреждающее сообщение. Рассмотрим пример,

```
public void m()  {
    char a, b;

    // ...

    if(a==b) {
       Console.WriteLine("Переменные a и b равны.");
       return;
    } else {
       Console.WriteLine("Переменные a и b равны.");
       return;
    }
    Console.WriteLine("Эта строка кода никогда не будет выполнена.");
}
```

Здесь возврат управления из метода m() всегда будет происходить до выполнения последнего оператора WriteLine();. Если вы попытаетесь скомпилировать этот метод, будет выдано сообщение, что в программе обнаружен код, который никогда не будет выполнен. Появление в программе невыполнимого кода — это результат ошибки программиста, поэтому к таким сообщениям всегда следует относиться серьезно.

Использование параметров

При вызове метода ему можно передавать одно или несколько значений. Как уже говорилось, передаваемое методу значение называется *аргументом*. Переменная внутри метода, которая принимает аргумент, называется *параметром*. Параметры объявляются внутри круглых скобок, следующих за именем метода. Синтаксис объявления параметров аналогичен синтаксису объявления переменных. Параметр находятся в области видимости своего метода. Назначение параметра — при выполнении метода принимать значение аргумента, но в целом он действует, как любая другая локальная переменная.

Далее приведена простая программа, в которой для передачи методу значения используется параметр. Внутри класса `CkhNum` метод `isEven()` возвращает значение `true`, если передаваемое ему значение является четным. В противном случае он возвращает значение `false`. То есть метод `isEven()` возвращает значения типа `bool`.

```
// Простая программа, в которой демонстрируется использование параметра для
// передачи методу значения.

using System;

class ChkNum {
  // Определение метода, который возвращает значение типа bool.
  public bool isEven (int x) {
    if ( (x%2) == 0) return true;
    else return false;
  }

}
class ParmDemo {
  public static void Main() {
    ChkNum e = new ChkNum();

    if(e.isEven(10)) Console.WriteLine("Число 10 четное.");

    if(e.isEven(9)) Console.WriteLine("Число 9 четное.");

    if(e.isEven(8)) Console.WriteLine("Число 8 четное.");

  }
}
```

Переменная x является параметром метода isEven () и имеет тип int.

В результате выполнения этой программы будут выведены следующие строки:

```
Число 10 четное.
Число 8 четное.
```

Метод `isEven()` вызывается три раза и каждый раз ему передается новое значение. Рассмотрим этот процесс более подробно, при этом обратим внимание на способ вызова метода `isEven()` — аргумент указывается внутри круглых скобок. Когда метод `isEven()` вызывается первый раз, ему передается значение 10 (при выполнении метода параметр `x` будет иметь значение 10). Во втором вызове аргументом является значение 9 (параметр `x` будет иметь значение 9). При третьем вызове параметр `x` получает значение 8. То есть параметр метода, переменная `x`, получает значение, которое передается как аргумент во время вызова метода `isEven()`.

Метод может иметь несколько параметром. Объявляемые параметры разделяются запятыми. Например, класс `Factor` содержит метод `isFactor()`, в котором определяется, является ли первый параметр делителем второго параметра.

```
using System;

class Factor {
  public bool isFactor(int a, int b) {
    if( (b % a) == 0) return true;
    else return false;
   }
}

class IsFact (
  public static void Main()   {
    Factor x = new Factor!);

    if(x.isFactor(2, 20))
      Console.WriteLine("Число  2  является  делителем  числа  20.");
    if (x.isFactor(3, 20))
      Console.WriteLine("Эта строка не будет выведена на экран.");

  }
}
```

Этот метод имеет два параметра.

Обратите внимание, что при вызове метода `isFacuor()` аргументы тоже разделены запятыми. Используемые параметры могут иметь различные типы. Например, такой фрагмент кода

```
int myMeth(int a, double b, float c) {
// ...
```

не является ошибочным.

Дальнейшее усовершенствование класса `Vehicle`

Можно продолжить усовершенствование класса `Vehicle` и добавить возможность вычисления количества топлива, необходимого автомобилю дня преодоления заданного расстояния. Новый метод, который будет иметь имя `fuelneeded()`, принимает в качестве аргумента значение расстояния, которое необходимо преодолеть автомобилю, и возвращает значение необходимого количества топлива (в галлонах). Приведем определение метода `fuelneeded()`:

```
public double fuelneeded(int miles) {

  return (double) miles / mpg;

}
```

Обратите внимание, что метод возвращает значение типа `double`. Это весьма уместно, поскольку значение необходимого количества топлива не обязательно будет целочисленным.

Далее представлен весь код класса `Vehicle`, в котором определен метод `fuelneeded()`.

```
/*
  Для вычисления количества топлива, необходимого автомобилю, чтобы преодолеть
  указанное расстояние, в программе используется метод с параметром.
 */
```

```
using System;

class Vehicle {
  public int  passengers;    // Количество пассажиров.
  public int  fuelcap;       // Емкость топливного бака (в галлонах).
  public int  mpg;           // Расстояние (в милях), которое данный
                             //  автомобиль  может  проехать,  используя  один
                             // галлон топлива.

     // Метод возвращает значение максимального расстояния, которое может
  // проехать автомобиль с полным топливным баком.
  public int range() {
    return mpg * fuelcap;
  }

  // Метод вычисляет количество топлива, требуемое для преодоления
     расстояния, значение которого передается методу в качестве параметра.
  public double fuelneeded(int miles) {
    return (double) miles / mpg;
  }
}

class CompFuel {
  public static void Main()     {
    Vehicle minivan = new Vehicle();
    Vehicle  sportscar  =  new  Vehicle();
    double  gallons;
    int dist = 252;

    //  Значения  присваиваются  переменным  объекта  minivan.
    minivan.passengers  =  7;
    ninivan.fuelcap  =  16;
    minivan.mpg = 21;

    //  Значения  присваиваются  переменным  объекта  sportscar.
    sportscar.passengers  =  2;
    sportscar.fuelcap  =  14;
    sportscar.mpg = 12;

    gallons = minivan.fuelneeded(dist);

    Console.WriteLlne("Чтобы  проехать  "  +  dist  +  "  мили,  микроавтобусу  "  +
                      "требуется " + gallons + " галлонов топлива.");

    gallons = sportscar.fuelneeded(dist);

    Console.WriteLine("Чтобы проехать " + dist + " мили, спортивному " +
                      "автомобилю требуется " + gallons + " галлон топлива.");

  }
}
```

Ниже представлен результат выполнения этой программы.

```
Чтобы проехать 252 мили, микроавтобусу требуется 12.0 галлонов топлива.
Чтобы проехать 252 мили, спортивному автомобилю требуется 21.0 галлон топлива.
```

Минутный практикум

1. В каких случаях для доступа к переменной экземпляра или к методу нужно указывать имя объекта и использовать оператор точка (.)? В каких случаях имя переменной или метода может использоваться непосредственно?
2. Объясните разницу между аргументом и параметром.
3. Назовите два способа возврата управления из метода.

Проект 4-1. Создание справочного класса

HelpClassDemo.cs

Классы необходимы большой программе в качестве строительных блоков. Для этого каждый класс должен представлять собой единый (законченный) функциональный блок, выполняющий четко определенные действия. Классы должны быть небольшими насколько это возможно, но в то же время и не слишком маленькими. Потому что классы, для которых заданы некоторые второстепенные функции, запутывают и деструктурируют код, а классы, которые выполняют недостаточно функций, не предоставляют программе необходимую функциональность. Золотая середина находится там, где наука о программировании переходит в искусство программирования. Поэтому чем больше у программиста опыта, тем быстрее он находит оптимальное решение.

Попробуем для тренировки преобразовать справочную систему из проекта 3-3 (см. предыдущую главу) в справочный класс. Сначала рассмотрим, чем хороша эта идея. Во-первых, справочная система представляет один логический блок. Поскольку задачей является вывод на экран синтаксиса операторов C#, выполняемая функция блока компактна и четко определена. Во-вторых, помещение справочной системы в класс облегчает работу программиста — если в какой-либо программе понадобится предложить пользователю справочную систему, то достаточно будет создать объект, являющийся экземпляром данного класса. Наконец, поскольку справка инкапсулирована, ее можно обновить или дополнить, причем это не приведет к побочным эффектам в программе, использующей справочную систему.

Пошаговая инструкция

1. Создайте новый файл и назовите его `HelpclassDemo.cs`. Чтобы не вводить код заново, вы можете скопировать код программы из файла `Help3.cs` в новый файл `HelpClassDemo.cs`.

---

1. Для доступа к переменной экземпляра из кода, не являющегося частью класса, в котором определена эта переменная, необходимо указать имя объекта, оператор точка (.) и имя переменной. Для доступа к переменной экземпляра из кода, являющегося частью класса, в котором определена данная переменная, можно непосредственно указать имя переменной. Эти же правила справедливы для методов.
2. Аргумент — это значение, которое передается методу при его вызове. Параметром является определенная методом переменная, которая принимает значение аргумента.
3. Возврат управления из метода может осуществляться с помощью оператора `return`. Если метод имеет тип `void`, то возврат управления в то место программы, из которого метод был вызван, осуществляется при достижении закрывающей фигурной скобки. Методы, возвращающие значение какого-либо типа (не-`void`), должны использовать оператор `return` для возврата значения указанного типа, то есть возврат управления из метода при достижении закрывающей фигурной скобки невозможен.

2 Для создания класса нужно точно определить составляющие справочной системы. Например, в проекте `Help3.cs` определен код для вывода на экран меню, ввода пользователем запроса, проверки корректности запроса и вывода на экран информации о выбранном элементе меню. Программа продолжает выполнять цикл до тех пор, пока пользователь не введет символ `q`. Очевидно, что вывод меню, проверка правильности ввода и вывод информации являются неотъемлемыми частями справочной системы. А получение программой пользовательского ввода и определение ситуаций, когда необходимо обрабатывать повторные запросы пользователя, — это вопросы, которые не являются неотъемлемыми компонентами справочной системы. Следовательно, в нашу задачу входит создание класса, выводящего меню, проверяющего корректность запроса и выводящего запрашиваемую информацию, назовем эти методы `showmenu()`, `isvalia()` и `helpon()` соответственно.

3. Создайте метод `helpon()`, как показано ниже:

```
public void helpon(char what) {
  switch(what) (
    case '1':
      Console.WriteLine("Оператор if:\n");
      Console.WriteLine("if(condition) statement;");
      Console.WriteLine("else     statement;");
      break;
    case '2':
      Console.WriteLine("Оператор switch:\n");
      Console.WriteLine("switch(expression) {") ;
      Console.WriteLine(" case constant:");
      Console.WriteLine("      statement sequence”);
      Console.WriteLine("      break;");
      Console.WriteLine("     // ...");
      Console.WriteLine(")");
      break;
    case '3':
      Console.WriteLine("Цикл for:\n”);
      Console.Write("for(init; condition; iteration)");
      Console.WriteLine("     statement;");
      break;
    case '4':
      Console.WriteLine("Цикл while:\n");
      Console.WriteLine("while(condition)       statement;");
      break;
    case '5':
      Console.WriteLine("Цикл do-while:\n");
      Console.WriteLine("do {");
      Console.WriteLine(" statement;");
      Console.WriteLine("}    while    (condition);");
      break;
    case '6':
      Console.WriteLine("Оператор break:\n");
      Console.WriteLine("break;   or   break   label;");
      break;
    case '7':
      Console.WriteLine("Оператор continue:\n");
      Console.WriteLine("continue;   or   continue   label;");
      break;
    case '8':
      Console.WriteLine("Оператор goto:\n");
```

```
        Console.WriteLine("goto label;");
        break;
    }
    Console.WriteLine();
}
```

4. Создайте метод `showmenu()` следующим образом:

```
public void showmenu() {
   Console.WriteLine("Справка по синтаксису: ");
   Console.WriteLine(" 1. Оператор if");
   Console.WriteLine(" 2. Оператор switch"};
   Console.WriteLine(" 3. Цикл for");
   Console.WriteLine(" 4. Цикл while");
   Console.WriteLine(" 5. Цикл do—while");
   Console.WriteLine(" 6. Оператор break");
   Console.WriteLine(" 7. Оператор continue");
   Console.WriteLine(" 8. Оператор goto\n");
   Console.WriteLine{"Введите порядковый номер оператора или цикла: ");
   Console.WriteLine("Для завершения работы программы введите символ q: ")
}
```

5. Создайте метод `isvalid()`, как показано ниже:

```
public bool isvaiid(char ch) {
   if (ch <  '1' |  ch >  '8'  & ch != 'q') return false;
   else return true;
}
```

6. Создайте из этих трех методов класс `Help`:

```
class Help {
  public void helpon(char what) {
     switch(what) {
       case '1':
         Console.WriteLine("Оператор if:\n");
         Console.WriteLine("if(condition) statement;");
         Console.WriteLine("else statement;"):
         break;
       case '2':
         Console.WriteLine("Оператор switch:\n");
         Console.WriteLine("switch(expression) {");
         Console.WriteLine(" case constant:");
         Console.WriteLine(" statement sequence");
         Console.WriteLine(" break;");
         Console.WriteLine (" // ...");
         Console.WriteLine("}");
         break;
       case '3':
         Console.WriteLine("Цикл for:\n”);
         Console.Write("for(init; condition; iteration)”);
         Console.WriteLine(" statement;") ;
         break;
       case '4':
         Console.WriteLine("Цикл while:\n");
         Console.WriteLine("while(condition) statoment;”);
         break;
       case 'b':
         Console.WriteLine("Цикл do-while:\n");
```

```
          Console.WriteLine("do {");
          Console.WriteLine(" statement;");
          Console.WriteLine("} while (condition);");
          break;
        case '6':
          Console.WriteLine("Оператор break:\n");
          Console.WriteLine("break; or break label;");
          break;
        case '7';
          Console.WriteLine("Оператор continue:\n");
          Console.Wr1teLine("continue; or continue label;");
          break;
        case '8':
          Console.WriteLine("Оператор goto:\n");
          Console.WriteLine("goto label;");
          break;
      }
      Console.WriteLine ();
    }
    public void shownenu() {
      Console.WriteLine("Справка по синтаксису: ");
      Console.WriteLine(" 1. Оператор if");
      Console.WriteLinet" 2. Оператор switch");
      Console.WriteLine(" 3. Цикл for");
      Console.WriteLine(" 4. Цикл while");
      Console.WriteLine(" 5. Цикл do-while”);
      Console.WriteLine(" 6. Оператор break");
      Console.WriteLine(" 7. Оператор continue");
      Console.WriteLine(" 8. Оператор goto\n");
      Console.WriteLine("Введите порядковый номер оператора или цикла: "):
      Console.WriteLine("Для завершения работы программы введите символ q: ");
    }

    public bool isvaiid(char ch) (
      if(ch < '1'   |  ch > '8'  & ch     'q') return false;
      else return true;
    }
  }
}
```

7. Теперь необходимо переписать метод `Main()` из проекта 3-3 таким образом, чтобы в нем использовался новый класс `Help`. Назовем главный класс (то есть класс, в котором находится метод `Main()`) `HelpClassDemo.cs`. Полный листинг программы `HelpClassDemo.cs` представлен ниже.

```
/* Проект 4-1.

   На основе справочной системы из проекта 3-3 создан класс Help.
*/

using System;

class Help {
  public void heipon(char what) {
```

```
    switch(what) {
      case '1':
        Console.WriteLine("Оператор if:\n");
        Console.WriteLine("if(condition) statement;");
        Console.WriteLine("else statement;");
        break;
      case '2':
        Console.WriteLine("Оператор switch:\n");
        Console.WriteLine("switch(expression) {");
        Console.WriteLine(" case constant:");
        Console.WriteLine("   statement sequence");
        Console.WriteLine("   break;");
        Console.WriteLine("  // ...");
        Console.WriteLine(")");
        break;
      case '3':
        Console.WriteLine("Цикл for:\n");
        Console.Write("for (init; condition; iteration)");
        Console.WriteLine(" statement;");
        break;
      case '4';
        Console.WriteLine("Цикл while:\n");
        Console.WriteLine("while(condition) statement;");
        break;
      case '5':
        Console.WriteLine("Цикл do-while:\n") ;
        Console -WriteLine("do {");
        Console.WriteLine(" statement;");
        Console.WriteLine ("} while (condition);");
        break;
      case '6':
        Console.WriteLine("Оператор break:\n") ;
        Console.WriteLine("break; or break label;");
        break;
      case '7':
        Console.WriteLine("Оператор continue;\n");
        Console.WriteLine("continue; or continue label;");
        break;
      case '8':
        Console.WriteLine("Оператор goto:\n");
        Console.WriteLine("goto label;");
        break;
    }
   Console.WriteLine();
 }

public void showmenul) {
   Console.WriteLine("Справка по синтаксису: ");
   Console.WriteLine(" 1. Оператор if") ;
   Console.WriteLine(" 2. Оператор switch");
   Console.WriteLine(" 3. Цикл for");
   Console.WriteLine(" 4. Цикл while");
   Console.WriteLine(" 5. Цикл do—while");
   Console.WriteLine(" 6. Оператор break");
   Console.WriteLine(" 7. Оператор continue");
   Console.WriteLine(" 8. Оператор goto\n");
```

```
      Console.WriteLine ("Введите порядковый номер оператора или цикла: ");
      Console.WriteLine ("Для завершения работы программы введите символ q: ");
    }

    public bool isvalid(char ch) {
      if (ch < '1' | ch > '8' & ch != 'q') return false;
      else return true;
    }

  }

  class HelpClassDemo {
    public static void Main() {
      char choice;
      Help hlpobj = new Help();

      for(;;) {
        do {
          hlpobj.shownenu();
          do {
            choice = (char) Console.Read();
          } while(choice == '\n' | choice == '\r');

        } while( !hlpobj.isvalid(choice) );

        if (choice == 'q') break;

        Console.WriteLine("\n");


        hlpobj.helpon(choice);
      }
    }
  }
```

Тестирование данной программы показывает, что выполняемые ею функции остались прежними. Преимущество же нового подхода состоит в том, что теперь в распоряжении программиста есть справочная система, которая может исполь-зоваться повторно, когда в этом возникнет необходимость.

Конструкторы

В предыдущих примерах переменным экземпляра каждого объекта класса `vehicle` необходимо было присваивать значения вручную, используя следующую последовательность операторов:

```
minivan.passengers = 7;
minivan.fuelcap = 16;
minivan.mpg = 21;
```

Такой способ никогда не используется в профессионально написанных программах C#, потому что, во-первых, при ручном вводе всегда существует вероятность случайной ошибки (вы можете забыть присвоить значение одной из переменных), во-вторых, в C# предусмотрен более эффективный способ выполнения этой задачи — использование конструктора.

Конструктор класса инициализирует объект при его создании. Он имеет то же имя, что и его класс, и синтаксически похож на метод. Однако в конструкторах тип возвращаемого значения не указывается явно. Общий синтаксис конструктора:

```
class-name() {
    // код конструктора
}
```

Как правило, конструкторы используются для присваивания начальных значений переменным экземпляра, определенным классом, или для выполнения любых других процедур инициализации, необходимых для создании полностью сформированного объекта.

Все классы имеют конструкторы независимо от того, определен он или нет. По умолчанию в C# предусмотрено наличие конструктора, который присваивает нулевые значения всем переменным экземпляра (для переменных обычных типов) и значения `null` (для переменных ссылочного типа). Но если конструктор явно определен в классе, то конструктор по умолчанию использоваться не будет.

Рассмотрим простой пример:

```
// Пример простого конструктора.
using System;

  class MyClass {
    public int x;

  public MyClass()   {
     x = 10;
   }
}

class ConsDemo {
  public static void Main() {
     MyCiass t1 = new MyCiass();
     MyCiass t2 = new MyCiass();

     Console.WriteLine(tl.x + ” " + t2.x);
   }
}
```

Конструктор класса `MyClass`.

В этом примере используется конструктор класса MyClass

```
public MyClass()   {
  x = 10;
}
```

Отметим, что конструктор определен как `public`. Это сделано потому, что конструктор будет вызываться из кода, определенного за пределами его класса. Конструктор присваивает переменной экземпляра `x` класса `MyCiass` значение 10. Этот конструктор вызывается оператором `new` при создании объекта. Например, в строке кода

```
MyClass tl = new MyCiass();
```

конструктор `MyCiass()` вызывается для создания объекта `t1`, присваивая переменной `t1.x` значение `10`. То же происходит и для объекта `t2` — после вызова конструктора переменная `t2.x` также получает значение 10. Таким образом, в результате выполнения программы будут выведены следующие значения:

```
10 10
```

Конструкторы с параметрами

В предыдущем примере был использован конструктор без параметров. Иногда это приемлемо, но чаще применяется конструктор с одним или несколькими параметрами. Параметры используются в конструкторе так же, как в методе. В следующем примере представлен класс `MyClass`, содержащий конструктор с параметрами.

```
// Использование конструктора с параметрами,

using System;

class MyClass {
  public int x;

  public MyClass(int i) {
    x = i;
  }
}

class ParmConsDemo {
  public static void Main() {
    MyClass t1 = new MyClass(10);
    MyClass t2 = new MyClass(88);

    Console.WriteLine(t1.x + " " + t2.x);
  }
}
```

Ниже показан результат выполнения этой программы.

```
10 88
```

В данной версии класса `MyClass` конструктор `MyClass()` имеет один параметр (переменную `i` типа `int`), который используется для присваивания начального значения переменной экземпляра `x`. Следовательно, после выполнения оператора

```
MyClass t1 = new MyClass(10);
```

значение 10 передается параметру `i`, а затем оно присваивается переменной `x`.

Добавление конструктора к классу Vehicle

Можно усовершенствовать класс `Vehicle`, добавив к нему конструктор, который при создании объекта будет автоматически инициализировать переменные `passengers`, `fuelcap` и `mpg`. Обратите особое внимание на строки кода, в которых создаются объекты класса `Vehicle`.

```
// В программе используется новая версия класса Vehicle, в которой определен
// конструктор.

using System;

class Vehicle {
  public int passengers; // Количество пассажиров.
  public int fuelcap;    // Емкость топливного бака (в галлонах).
  public int mpg;        // Расстояние (в милях), которое данный автомобиль
                         // может проехать, используя один галлон топлива.
  // Конструктор класса Vehicle,
  public Vehicle(int p, int f, int m) {
    passengers = p;
```

Конструктор класса `Vehicle`.

```
    fuelcap = f;
    mpg = m;
  }

  // Метод возвращает значение максимального расстояния, которое может
  // проехать автомобиль с полным топливным баком,
  public int range() {
    return mpg * fuelcap;
  }

  // Метод вычисляет количество топлива, необходимое для преодоления
  // расстояния, значение которого передается методу в качестве параметра,
  public double fuelneeded(int miles) {
    return (double) miles / mpg;
  }
}
class VehConsDemo {
  public static void Main() {

    // Создание двух объектов класса Vehicle.
    Vehicle minivan = new Vehicle(7, 16, 21);
    Vehicly sportscar = new Vehicle(2, 14, 12);
    double gallons;
    int dist = 252;

    gallons = minivan.fuelneeded(dist);

    Console.WriteLine("Чтобы проехать " + dist + " мили, микроавтобусу " +
                      "требуется " + gallons + " галлонов топлива.");

    gallons = sportscar.fuelneeded(disc);

    Console.WriteLine("Чтобы проехать " + dist + " мили, спортивному " +
                      "автомобилю требуется " + gallons + " галлон топлива.");

  }
}
```

Передача характеристик автомобиля объекту класса `Vehicle` с использованием его конструктора.

При создании объектов `minivan` и `sportscar` конструктор `Vehicle()` присваивает переменным этих объектов начальные значения. Каждый объект инициализируется указываемыми аргументами. Например, в строке кода

```
Vehicle minivan = new Vehicle (7, 16, 21);
```

значения 7, 16 и 21 передаются конструктору при создании объекта (выполнении оператора `new`). Следовательно, копии переменных `passengers`, `fuelcap` и `mpg` объекта `minivan` будут содержать значения 7, 16 и 21 соответственно, а результат выполнения этой программы будет таким же, как в предыдущей версии.

Минутный практикум

1. Что такое конструктор? Когда он выполняется?
2. Указывается ли в конструкторе тип возвращаемого значения?

---

1. Конструктором называется метол, используемый для инициализации создаваемого объекта. Конструктор выполняется при создании объекта его класса.
2. Нет.

Оператор new

Теперь, когда мы уже достаточно знаем о классах и их конструкторах, подробно рассмотрим оператор new. Этот оператор имеет следующий синтаксис:

```
class-var = new class-name();
```

Здесь словосочетание `class-var` — создаваемая переменная, тип которой (`class`) указывается перед именем переменной, а словосочетание `class-name` — это имя класса, экземпляр которого создается. (Можно сказать, что вместо подстановочных слов `class` и `class-name` указывается одно и то же имя класса, экземпляр которого создается.) Имя класса, за которым следует пара круглых скобок, является конструктором класса. Если класс не имеет своего конструктора, оператор `new` будет использовать конструктор по умолчанию, предоставленный C#. Следовательно, оператор `new` может использоваться для создания объекта любого класса.

Поскольку размер памяти ограничен, существует вероятность, что оператор `new` не сможет выделить для объекта нужное количество памяти. Если это произойдет, то возникнет исключительная ситуация выполнения программы. (Как обрабатывать исключительные ситуации, вы узнаете из главы 9.) При выполнении программ этой книги вам не нужно беспокоиться о недостаточном объеме памяти, но в будущем при создании своих программ всегда учитывайте вероятность этого.

Ответы профессионала

Вопрос. Почему отсутствует необходимость использования оператора `new` для создания переменных обычного типа (таких, как `int` или `float`)?

Ответ. В C# переменная обычного типа сама хранит свое значение. Память для хранения этой переменной автоматически выделяется при компилировании программы, и нет необходимости явно выделять память с помощью оператора `new`. Что касается переменных ссылочного типа, то они хранят только ссылку на объект. Память для хранения объекта выделяется динамически (во время выполнения программы).

Превращение переменных обычного типа в переменные ссылочного типа может заметно снизить производительность программы. При использовании переменных ссылочного типа происходят непрямые обращения к значениям, что снижает скорость выполнения программы.

Для эксперимента можно применить оператор `new`, чтобы создать переменную обычного типа

```
int i = new int();
```

При этом будет вызван конструктор по умолчанию для типа `int`, который присвоит переменной `i` начальное значение `0`. Однако следует заметить, что при этом память не будет выделена динамически. Большинство программистов не используют оператор `new` для создания переменных обычного типа.

Деструкторы и «сборка мусора»

Вы уже знаете, что при использовании оператора new память для объектов выделяется динамически. Но поскольку объем памяти ограничен, существует возможность неудачной попытки создания оператором new необходимого объекта. Поэтому один

из ключевых компонентов схемы динамического выделения памяти — механизм освобождения памяти, занимаемой неиспользуемыми объектами, лля того чтобы ее можно было выделить пол вновь создаваемые объекты. Во многих языках программирования освобождение ранее выделенной памяти осуществляется вручную (например, в C++для освобождения памяти используется оператор dele-?). Cз располагает для этого более эффективным механизмом, называемым *«сборкой мусора».*

Когда в C#-про грамме отсутствуют обращения к объекту, то он рассматривается как не использующийся более, и система автоматически без участия программиста освобождает занимаемую им память, которая в дальнейшем может быть выделена под новые объекты.

«Сборка мусора» периодически осуществляется в ходе всей программы, причем для начала этой операции должны выполняться два условия — во-первых, наличие объектов, которые можно удалить из памяти, а во-вторых, необходимость такой операции. Поскольку процесс «сборки мусора» занимает некоторое время, система исполнения программы осуществляет ее только в случае необходимости, причем программист не может точно определить момент, когда это произойдет.

Деструкторы

В C# предусмотрена возможность создания метода, который вызывается непосредственно перед удалением объекта из памяти при помощи операции «сборки мусора». Этот метод называется *деструктором* и может использоваться для гарантии корректного удаления объекта из памяти. Например, с помощью деструктора можно удостовериться, что файл, к которому имело место обращение из данного объекта, будет закрыт. Деструктор имеет следующий синтаксис:

```
~class-name()   {
   // код деструктора
}
```

Здесь словосочетание `class-name` — это имя класса. То есть деструктор объявляется так же, как конструктор, за исключением того, что в деструкторе перед именем класса используется символ тильда (~). Отметим, что при определении деструктора не указывается тип возвращаемого значения.

Для добавления деструктора к классу нужно просто добавить его определение к коду этого класса (то есть деструктор является обычным членом класса). Внутри деструктора назначаются действия, которые должны быть выполнены перед возвращением памяти, выделенной для этого объекта. Деструктор вызывается во всех случаях, когда объект его класса должен быть удален из памяти.

Важно понимать, что деструктор вызывается непосредственно перед выполнением операции «сборки мусора». (Он не вызывается, например, когда объект класса выходит из области видимости. Этим деструкторы в C# отличаются от деструкторов в C++, где они *вызываются*, когда объект выходит за пределы области видимости, в которой этот объект был создан.) Это означает, что невозможно точно определить, когда будет выполняться код деструктора. Кроме тою, программа может завершить работу до начала операции «сборки мусора», в этом случае деструктор не будет вызываться вообще.

Проект 4-2. Демонстрация работы деструкторов

DestructDemo.cs

Мы уже говорили, что объекты не обязательно удаляются из памяти, если они больше не попользуются в программе. «Сборка мусора» производится только тогда, когда она может быть выполнена эффективно (обычно для нескольких объектов одновременно). Следовательно, для демонстрации работы деструктора необходимо создать и уничтожить большое количество объектов. Именно этому мы посвятим данный проект.

Пошаговая инструкция

1. Создайте новый файл и назовите его `Destruct.cs`.
2. Создайте класс `Destruct`, как показано ниже:

```
class Destruct {
    int x;

  public Destruct(int i) {
    x = i;
  }

  // Вызывается при удалении объекта из памяти.
  ~Destruct() {
    Console.Writebine("Удаление из памяти объекта " + x);
  }

  // Метод создает объект, который сразу будет уничтожен.
  public void generator(int i) {
    Destruct o = new Destruct(i);
  }

}
```

Конструктор присваивает переменной экземпляра `x` значение, передаваемое конструктору в качестве аргумента. В этой программе деструктор с помощью переменной `x` выводит на экран порядковый номер уничтожаемого объекта. Особое внимание обратите на метод `generator()`, который создаст и затем сразу уничтожает объект типа `Destruct`. В следующем пункте инструкции вы увидите, как это осуществляется.

3. Создайте класс `DestructDemo`, как показано ниже:

```
class DestructDemo {
  public static void Main()   {
    int count;

    Destruct ob = new Destruct(0);

    /* Далее с помощью цикла for создается большое количество объектов.
       После использования всей свободной памяти выполняется операция
       «сборка мусора». Необходимое для этого количество объектов зависит
       от объема установленной на компьютере памяти, поэтому, возможно,
       придется увеличить число, используемое в условном выражении цикла
       (количество проходов цикла).
     */
```

```
    for (count=1; count < 100000; count++)
      ob.generator(count);
  }
}
```

Вначале в этом классе создастся объект `ob` типа `Destruct`. Затем с помощью этого объекта и вызова определенного в нем метода `generator()` создается 100000 объектов. В результате на некотором этапе выполнения программы объекты станут не только создаваться, но и уничтожаться, причем на различных этапах будут происходить операции «сборки мусора». Как часто и когда именно это будет происходить, зависит от нескольких факторов — начального объема свободной памяти, типа операционной системы и так далее. Но с определенного момента на экране начнут появляться сообщения, генерируемые деструктором. Если такие сообщения не выводятся, попробуйте увеличить число создаваемых объектов путем увеличения значения переменной `count` в цикле `for`.

4. Ниже представлен полный листинг программы `DectructDemo.cs`:

```
/*
   Проект 4-2.
   В этой программе демонстрируется работа деструктора.
*/

using System;

class Destruct {
  public int x;

  public Destruct(int i) {
    x = i;
  }

  // Вызывается при удалении объекта из памяти.
  ~Destruct() {
    Console.WriteLine("Destructing " + x);
  }

  // Метод создает объект, который сразу уничтожается,
  public void generator(int i) {
    Destruct o = new Destruct(i);
  }

}

class DestructDemo {
  public static void Main()    {
    int count:;

    Destruct ob = new Destruct(0);

    /*  Далее с помощью цикла for создается большое количество объектов.
        После использования всей свободной памяти выполняется операция
        «сборки мусора». Необходимое для этого количество объектов зависит
        от объема установленной на компьютере памяти, поэтому, возможно,
        придется увеличить число, используемое в условном выражении цикла
         (количество проходов цикла).
     */
```

```
      for (count=1; count < 100000; count++)
        ob.generator(count);
    }
  }
```

Ключевое слово this

В завершение этой главы рассмотрим особенности использования ключевого слова this. При вызове метода ему автоматически передается неявным аргумент, который является ссылкой на вызывающий объект (то есть на объект, с данными которого будет работать метод). Эта ссылка называется this. Чтобы понять, как работает ссылка this. рассмотрим программу, в которой создается класс Pwr, предназначенный для вычисления результата возведения числа в некоторую целочисленную степень.

```
using System;

class Pwr {
  public double b;
  public int e;
  public double val;

  public Pwr (double num, int exp) {
    b = num;
    e = exp;

    val = 1;
    if (exp==0) return;
    for ( ; exp>0;  exp--) val = val * b;
}

  public double get_pwr() {
    return val;
 }
}

class DemoPwr {
  Pablic static void Main() {
    Pwr x  =  new Pwr(4.0,  2);
    Pwr y  =  new Pwr(2.5,  1);
    Pwr z  =  new Pwr(5.7,  0);

    Console.WriteLine(x.b + " в " + x.e + "-й степени = " + x.get_pwr());
    Console.WriteLine(y.b + " в " + y.e + "-й степени = " + y.get_pwr());
    Console.WriteLine(z.b + " в " + z.e + "-й степени = " + z.got_pwr());
  }
}
```

Как вы знаете, в пределах метода доступ к другим членам класса может осуществляться без указания имени объекта или класса, то есть непосредственно. Следовательно, внутри метода get_pwr при выполнении оператора

```
return val;
```

будет возвращена копия переменной val, ассоциированная с вызывающим объектом. По этот же оператор можно написать следующим образом:

```
return this.val;
```

Здесь ссылка this указывает на объект, для которого был вызван метод get_pwr(). Следовательно, переменная this.val является копией переменной val данного объекта. Например, если метод get_pwr() вызван для объекта x, то ссылка this предыдущем операторе указывает на объект x. Можно сказать, что запись оператора без использования ключевого слова this является сокращенной формой записи.

Ниже представлен весь класс Pwr, написанный с использованием ключевого слова this.

```
using System;

class Pwr {
  public  double  b;
  public  int  e;
  public double val;

  public  Pwr(double  num,  int  exp)  {
    this.b  =  num;
    this.e = exp;

    this, val = 1;
    if(exp==0) return;
    for{ ; exp>0; exp--) this.val = this.val * this.b;
  }

  public  double  get_pwr()  {
    return this.val;
  }
}
```

Скорее всего, ни один опытный программист на языке C# не напишет класс Pwr подобным способом, поскольку никакого преимущества такая форма записи не дает. Но в некоторых случаях ссылка this может быть полезна. Например, синтаксис C# позволяет использовать одинаковые имена для параметров или локальных переменных и для переменных экземпляра. Когда это происходит, локальное имя *скрывает* переменную экземпляра, и доступ к ней можно получить, используя ссылку this. Например, приведенный ниже конструктор Pwr() является синтаксически действительным, хотя такой стиль применять не рекомендуется.

```
public  Pwr(double  b,  int  e)  {
    this.b  =  b;
    this.e = e;

    val = 1;
    if(e==0) return;
    for( ; e>0; e--) val = val * b;
}
```

В этой версии конструктора Pwr() имена параметров и имена переменных экземпляра одинаковы, поэтому переменные экземпляра будут скрыты. А если вы используете ссылку this, то можете «открыть» переменную экземпляра.

Контрольные вопросы

1. В чем состоит различие между классом и объектом?
2. Как определяется класс?
3. Собственные копии чего имеет каждый объект?
4. Покажите, как объявить объект с именем `counter` класса `MyCounter`, используя два отдельных оператора.
5. Запишите определение метода `myMeth()`, который возвращает значение типа `double` и имеет два параметра `a` и `b` типа `int`.
6. Как из метода возвращается управление, если метод возвращает значение?
7. Какое имя имеет конструктор?
8. Какие функции выполняет оператор `new`?
9. Что такое «сборка мусора»? Как эта операция выполняется? Что такое деструктор?
10. Объясните предназначение ключевого слова `this`?

Подробнее о типах данных и оператора

- Массивы
- Объекты типа `string`
- Цикл `foreach`
- Побитовые операторы
- Оператор ?

В этой главе мы вернемся к описанию типов данных и операторов C#, рассмотрим массивы, данные типа `string`, побитовые операторы и тернарный оператор `?`, а также подробно расскажем о цикле `foreach`.

Массивы

Массив — это коллекция переменных одного типа. При обращении к массиву указывается его имя и индекс элемента, которому было присвоено значение данного типа. Массивы могут быть как одномерными, так и многомерными, хотя чаще используются одномерные массивы. Массивы применяются для решения многих задач, поскольку предоставляют удобные средства группировки переменных одного типа. Например, вы можете использовать массив для хранения информации о среднесуточной температуре каждого дня месяца или для хранения комплекта символов либо коллекции объектов с данными о ваших однокурсниках, а также для других подобных целей.

Данные в массиве организованы так. что ими легко манипулировать, именно в этом заключается главное достоинство массива. Например, если массив хранит информацию об оценках, полученных абитуриентом, то путем обработки элементов массива в цикле легко вычислить средний балл экзаменуемого. Кроме того, эти данные легко сортируются.

В C# можно использовать массивы точно так же, как в других языках программирования, но здесь у них есть одна особенность. Она состоит в том, что массивы в C# реализованы как объекты, поэтому в данной книге сначала рассматриваются объекты, а затем массивы. Благодаря такой реализации массивов у C#-программ появляются дополнительные возможности, например, удаление из памяти неиспользуемых массивов при «сборке мусора».

Одномерные массивы

Одномерный массив — это список однотипных переменных. В программировании такие списки применяются достаточно часто. Например, одномерный массив можно использовать для хранения номеров счетов активных пользователей сети или дли хранения текущего рейтинга баскетбольной команды.

Для объявления одномерного массива применяется следующий синтаксис:

```
type() array—name = new type[size];
```

Здесь слово `type` указывает тип массива, то есть тип данных каждого элемента, из которых и состоит массив. Обратите внимание на квадратные скобки, которые следуют за ключевым словом, указывающим тип элементов массива. С помощью этих скобок объявляется одномерный массив (далее в этой главе будет показано, как объявляется двухмерный массив). Количество элементов, которые могут быть помещены в массив, определяется величиной `size`. Поскольку массивы реализованы как объекты, процесс создания массива состоит из двух этапов. На первом этапе объявляется имя массива — имя переменной ссылочного типа. На втором этапе для массива выделяется память и переменной массива присваивается ссылка на эту область памяти. То есть в C# с помощью оператора `new` память массиву выделяется динамически.

Примечание

Если у вас есть опыт программирования на C или C++, то обратите особое внимание на способ объявления массива в C#. В частности на то, что в C# квадратные скобки следуют за ключевым словом, указывающим тип, а не за именем массива.

Следующая строка кода создает массив типа `int`, состоящий из десяти элементов, ссылку на него присваивает переменной ссылочного типа `sample`:

```
int[] sample = new int[10];
```

Массив объявляется так же, как объект. Переменная `sample` хранит ссылку на адрес памяти, выделенной оператором `new`. Размер выделенной памяти достаточен хранения десяти элементов типа `int`.

Объявление массива (так же как объявление объекта) можно выполнить с помощью двух операторов. Например,

```
int[] sample;
sample = new int [10];
```

В этом случае после создания переменная массива `sample` не будет ссылаться на какой-либо физический объект. Только после выполнения второго оператора переменной присваивается ссылка на массив.

Доступ к отдельному элементу массива осуществляется с использованием *индекса:* С его помощью указывается позиция элемента в пределах массива. Первый элемент всех массивов в C# имеет нулевой индекс. Поскольку массив `sample` состоит из 10 элементов, значения индексов элементов этого массива находятся в диапазоне от 0 до 9. Для индексации массива необходимо указать количество его элементов в квадратных скобках. Таким образом, первый элемент массива `sample` указывается как `sample[0]`, а последний элемент — `sample[9]`. Например, в приведенной ниже программе в массив `sample` помещаются значения от 0 до 9.

```
// В программе демонстрируется использование одномерного массива.

using System;

class ArrayDemo {
  public static void Main()    {
    int[] sample = new int[10];
    int i;

    for(i = 0; i < 10; i = i+1)
      sample[i] = i;

    for (i = 0; i < 10; i = i+1)
      Console.WriteLine("Элементу sample[" + i + "] присвоено значение: " +
                        sample [i]);
  }
}
```

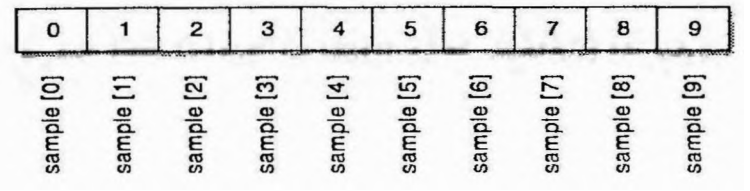
Первый элемент массив» имеет индекс 0.

В результате работы программы будут выведены следующие десять строк:

```
Элементу sample[0] было присвоено значение: 0
Элементу sample[1] было присвоено значение: 1
Элементу sample[2] было присвоено значение: 2
Элементу sample[3] было присвоено значение: 3
Элементу sample[4] было присвоено значение: 4
```

```
Элементу sample[5] было присвоено значение: 5
Элементу sample[6] было присвоено значение: 6
Элементу sample[7] было присвоено значение: 7
Элементу sample[8] было присвоено значение: 8
Элементу sample[9] было присвоено значение: 9
```

Схематически массив `sample` выглядит следующим образом:



Массивы широко используются в программировании, поскольку дают возможность легко управлять большим количеством однотипных переменных. Например, приведенная ниже программа с помощью обработки элементов массива `nums` в цикле `for` находит максимальное и минимальное значения элементов этого массива.

```
// Программа находит минимальное и максимальное значения элементов массива.

using System;

class MinMax {
  public static void Main() {
    int[] nums = new int[10];
    int min, max;

    nums[0] = 99;
    nums[1] = -10;
    nums[2] = 100123;
    nums[3] = 18;
    nums[4] = -978;
    nums[5] = 5623;
    nums[6] = 463;
    nums[7] = -9;
    nums[8] = 287;
    nums[9] = 49;

    min = max = nums[0];
    for(int i=1; i < 10; i++) {
      if(nums[i] < min) min = nums[i];
      if(nums[i] > max) max = nums[i];
    }
    Console.WriteLine("Минимальное значение: " + min + ", максимальное " +
                      "значение: " + max);
  }
}
```

Результат выполнения этой программы будет следующим:

```
Минимальное значение: -978, максимальное значение: 100123
```

Инициализация массива

В предыдущей программе значения элементам массива `nums` задавались вручную с Использованием десяти отдельных операторов присваивания. Существует более эффективный способ выполнения этой операции. Инициализировать массив можно

сразу при его создании. Общий синтаксис инициализации одномерного массива выглядит так:

```
type[] array-name = (val1, val2, val3, ..., valN);
```

Здесь начальные значения заданы величинами от `val1` до `valN`. Элементам массива значения присваиваются по очереди слева направо, начиная с элемента с индексом 0. В C# массиву автоматически выделяется объем памяти, достаточный для хранения указываемого количества значений, при этом нет необходимости явно использовать оператор `new`. Ниже показан более эффективный способ написания программы `MinMax`:

```
// В программе используется инициализация массива при его создании.

using System.;

class MinMax {
  public static void Main() {
    int[] nums = { 99, -10, 100123, 18, -978,
                   5623, 463, -9, 287, 49 };
    int min, max;

    min = max = nums[0];
    for {int i=1; i < 10; i++) {
      if(nums[i] < min) min = nums[i];
      if(nums[i] > max) max = nums[i];
    }
    Console. WriteLine ("Минимальное значение: " + min + ", максимальное " +
                        "значение:  " + max);
  }
}
```

Значения, присваиваемые элементам массива ири его создании.

Хотя в этом нет необходимости, вы можете в экспериментальных целях использовать для инициализации массива оператор `new`. Например, способ инициализации массива `nums`, показанным ниже, не противоречит правилам, но использование оператора `new` в данном случае вовсе не обязательно.

```
int[] nums = new int[] { 99, -10, 100123, 18, -978,
                         5623, 463, -9, 287, 49 };
```

Такой способ инициализации можно использовать для присваивания ссылки на новый массив уже существующей переменной ссылочного типа:

```
int[] nums;
nums = new int[] { 99, -10, 100123, 18, -978,
                   5623, 463, -9,287, 49 };
```

В этом случае переменная массива `nums` объявляется в первом операторе, а инициализируется во втором.

Жесткий контроль над граничными значениями индексов

В C# строго контролируется обращение к элементу массива. Следует указывать индекс, не превышающий граничные значения индексов этого массива, в противном случае возникнет ошибка выполнения программы. Чтобы убедиться в этом, протестируйте следующую программу, в которой специально указывается несуществующий индекс (то есть производится попытка обращения к несуществующему элементу).

```
// В программе производится попытка обращения к несуществующему элементу
   // массива.
using System;

class ArrayErr {
  public static void Mam() {
    int[] sample = new int[10];
    int i;

       // Согласно условному выражению цикла for переменная i может принять
       // значение, превышающее верхнее граничное значение индексов массива
       // sample.
    :cr(i = 0; i < 100; i = 1+1)
      sample[i]  = i;
  }
}
```

Превышено верхнее «раничнос значение индексов массива sample.

На определенном этапе выполнения программы переменной `i` присваивается значение 10. При попытке обращения к элементу с индексом 10 программа будет закрыта и будет сгенерирован объект-исключение `IndexOutOfRangeException` (подробнее об этом см. в главе 10).

Минутный практикум

1. С использованием чего осуществляется доступ к элементу массива?
2. Как объявить массив типа `char`, к котором содержится 10 элементов?
3. Справедливо ли утверждение, что во время выполнения C#-программы не осуществляется контроль над граничными значениями индексов массива?

Проект 5-1. Сортировка массива

`Bubble.cs`

Поскольку в одномерном массиве данные расположены в индексируемом линейном списке, они легко поддаются сортировке. Вероятно, вы знаете, что существует множество различных алгоритмов сортировки. Среди них можно выделить алгоритм быстрой сортировки, пузырьковый алгоритм и алгоритм сортировки методом Шелла. Наиболее известным, простым и понятным является *пузырьковый алгоритм.* Хотя он не очень эффективен (фактически неприемлем для сортировки больших массивов), но может с успехом применяться для сортировки небольших массивов.

Пошаговая инструкция

1. Создайте файл и назовите его `Bufctle.cs`.
2. Пузырьковый алгоритм сортировки получил имя в соответствии с принципом своей работы. Рассмотрим его подробно. Предположим, имеется массив, состоящий из 20

---

1. Доступ к элементу массива осуществляется с использованием индекса.
2. Массив типа `char`, содержащий 10 элементов, объявляется следующим образом:
   `char[] a = new char[10];`.
   3. Нет. В C# во время выполнения программы строго запрещено обращаться к несуществующим элементам массива.

элементов, которые нужно отсортировать в порядке возрастания. Сравниваются значении двух находящихся рядом элементов. Если значение второго элемента окажется меньше значения первого, то некоторой промежуточной переменной присваивается значение второго элемента, второму элементу присваивается значение первого элемента, а первому элементу присваивается значение промежуточной переменной. Можно сказать, что элементы меняются местами, причем элемент с наименьшим значением перемешается к левой границе массива. Это можно сравнить с перемещением пузырька воздуха в воде — из глубины к поверхности, поэтому алгоритм сортировки и получил такое название. Для того чтобы в массиве из 20 элементов сравнить все соседние элементы (первый со вторым, второй с третьим и так далее) необходимо выполнить 20 — | = 19 сравнений, а поскольку наименьшее значение может оказаться у последнего элемента (двадцатого), то для его перемещения в самое начало массива нужно выполнить 19 перемещений. То есть нужно организовать два цикла — внешний и внутренний, каждый из которых должен выполняться 19 раз.

Ниже приведен фрагмент кода, составляющий ядро пузырькового алгоритма сортировки. Массив, в котором сортируются элементы, называется `nums`.

```
// Это код пузырькового алгоритма сортировки.
for(a=1; a < size; a++)
 for (b=size-1; b >= a; b--) {
   if(nums[b-1] > nums[b]) {       // Если значение предыдущего элемента больше
                                   // значения последующего элемента, то они
     t = nums[b-1];                // "меняются местами",
     nums[b-1] = nums[b];
     nums[b] = t;
   }
 }
```

Чтобы вы хорошо усвоили материал, изложенный в этом абзаце, еще раз коротко повторим принцип работы пузырькового алгоритма. Алгоритм построен на двух циклах `for`. Значения стоящих рядом элементов массива проверяются во внутреннем цикле. Когда обнаруживается пара элементов, значения которых расположены не в порядке возрастания, они меняются местами. При каждом проходе элементы с наименьшим значением передвигаются в положение, соответствующее их значению. Внешний цикл повторяет этот процесс до тех пор, пока все элементы массива не будут отсортированы.

3. Далее приведен код всей программы `Bubble`.

```
/*
   Проект 5—1.
   В программе демонстрируется использование пузырькового алгоритма
   сортировки элементов массива.
 */
using System;

class Bubble {
  public static void Main() {
    int [] nums = { 99, -10, 100123, 18, -978,
                    5623, 463, -9, 287, 49 };
    int a, b, t;
    int size;

    size = 10;   // Количество элементов массива.
```

```
    // Отображение значений элементов первоначального массива.
    Console.Write("Значения элементов первоначального массива: \n");
    for(int i=0; i < size; i++)
      Console.Write(" " + nums[i]);
    Console.WriteLine();

    // Это код пузырькового алгоритма сортировки,
    for(a=1; a < size; a++)
     for(b=size-1; b >= a; b--) {
       if(nums[b-l] > nums[b]) {    // Если значение предыдущего элемента
                                    // больше значения последующего элемента,
       t = nums[b-l];               // то они "меняются местами".
       nums[b-1] = nuraslb);
       nums[b] = t;
        }
     }
    // Отображение значений элементов отсортированного массива.
    Console.Write("Значения элементов отсортированного массива: \n");
    for(int i=0; i < size; i++)
      Console.Write(" " + nums[i]);
    Console.WriteLine();
  }
}
```

Ниже показан результат выполнения этой программы.

```
Значения элементов первоначального массива:
99, -10, 100123, 18, -978, 5623, 463, -9, 287, 49
Значения элементов отсортированного массива:
-978, -10, -9, 18, 49, 99, 287, 463, 5623, 100123
```

Хотя сортировка данных в соответствии с пузырьковым алгоритмом вполне приемлема для небольших массивов, она становится неэффективной при работе с большими массивами. Наилучшим универсальным алгоритмом сортировки является алгоритм быстрой сортировки. Но он основан на тех свойствах языка C#, которые вам еще не знакомы, об этом алгоритме мы расскажем в главе 6.

Многомерные массивы

Одномерные массивы являются самыми распространенными в программировании, но и многомерные массивы встречаются не так уж редко. Многомерным называется массив, который имеет две или больше размерностей, а доступ к отдельному элементу осуществляется в нем с помощью указания двух и более индексов.

Двухмерные массивы

Самым простым из многомерных массивов является двухмерный массив. В нем Расположение конкретного элемента определяется двумя индексами. Двухмерный Массив можно рассматривать как информационную таблицу, один индекс которой указывает номер строки, а другой — номер столбца.

Двухмерный целочисленный массив `table` с размерностью 10x20 объявляется следующим образом:

```
int[,] table = new int[10, 20];
```

Заметьте, что при объявлении данного массива две указываемые размерности массива разделены запятой. В первой части оператора с помощью запятой в квадратных скобках `[,]` указывается, что создается переменная ссылочного типа для двухмерного массива. Выделение памяти для массива с помощью оператора `new` происходит при выполнении второй части объявления массива:

```
new int[10, 20]
```

При этом создается массив с размерностью 10x20. Обратите внимание, что значения размерности разделены запятой.

Для получения доступа к элементу двухмерного массива необходимо указать два индекса, разделив их запятой. Например, для присваивания элементу массива `table[3, 5]` значения 10 необходимо использовать следующий оператор:

```
table [3, 5] = 10;
```

Ниже представлена программа, в которой элементам двухмерного массива присваиваются значения от 1 до 12, а затем эти значения выводятся на экран.

```
// В программе демонстрируется использование двухмерного массива,
using System;

class TwoD {
  public static void Main()    {
    int t:, i;
    int[,j table = new int[3, 4];

    for(t=0; t < 3; ++t) {
      for(i=0; i < 4;   ++i){
        table[t, i] = (t*4 ) +i + 1;
        Console.Write(table[t,i] + " ");

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

Объявление двухмерного массива с размерностью 3x4.

В этой программе элемент `table[D, 0]` будет иметь знамение 1, элемент `table[0, 1]` — значение 2, элемент `table[0, 2]` — значение 3 и т. я. Элемент `table[2, 3]` будет иметь значение 12. Ниже приведено схематическое изображение массива.

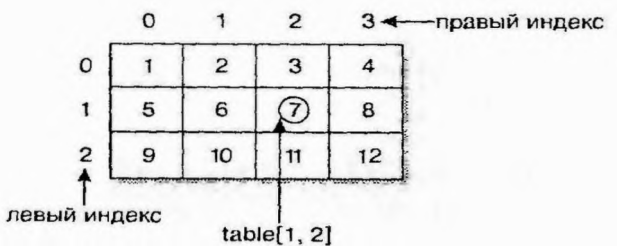


Рис. 5.1. Схематическое представление массива `table`

Примечание

Если у вас есть опыт программирования на C, C++ или Java, обратите внимание на то, что объявление или доступ к элементам многомерных массивов в C# осуществляется не так, как в этих языках. В них каждая размерность массива и индексы указываются в отдельных квадратных скобках, а в C# указываемые размерности разделяются запятыми.

Массивы, имеющие более двух размерностей

В C# разрешается использование массивов, имеющих более двух размерностей. Ниже представлен общий синтаксис объявления такого многомерного массива:

```
type[, ...,] name = new type[size1, size2, ..., sizeN];
```

Например, при выполнении следующего оператора будет создан целочисленный трехмерный массив с размерностью 4x10x3:

```
int[,,] multidim = new int[4, 10, 3];
```

Чтобы присвоить элементу массива `multidin[2, 4, 1]` значение `100`, необходимо выполнить следующий оператор:

```
multidin[2, 4, 1] = 100
```

Инициализация многомерных массивов

Инициализировать многомерный массив можно, поместив список значений каждой размерности в отдельный блок, заключенный в фигурные скобки. Например, ниже показан синтаксис инициализации двухмерного массива.

```
type[,] array_name = {
    {val, val, val,... val},
    {val, val, val,... val},
 .
 .
 .
    {val, val, val,... val},
};
```

Здесь слово `val` обозначает присваиваемое значение. Каждый внутренний блок присваивает значения элементам соответствующей строки. В пределах строки первое указываемое значение будет храниться в нулевом элементе строки, второе значение — в первом элементе и так далее. Обратите внимание, что блоки, содержащие значения, разделены запятыми, а за закрывающей фигурной скобкой блока со значениями первой размерности следует точка с запятой.

В приведенной ниже программе инициализируется массив `sqrs`, каждому элементу которого присваивается значение из диапазона1 - 10 и квадрат этого значения.

```
   // В программе демонстрируется инициализация двухмерного массива.
using System;

class Squares {
  public static void Main() {
    int[,] sqrs = {
      { 1, 1 },
      { 2, 4 },
```

Обратите внимание, что каждая строка имеет свой блок значений.

```
        { 3, 9 }
        { 4, 16 },
        { 5, 25 },
        { 6, 36 },
        { 7, 49 },
        { 8, 64 },
        { 9, 81 },
        { 10, 100 }

    int i, j;

    for(i=0; 1 < 10; i++) {
      for(j=0; j < 2; j++)
        Console.Write(sqrs[i, j] + " ");
      Console.WriteLine();
    }
  }
}
```

Ниже показан результат выполнения этой программы.

```
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
```

Невыровненные массивы

В предыдущей программе демонстрировалось создание двухмерного массива, который иначе можно назвать *прямоугольным массивом*. В табличном представлении прямоугольный массив — это двухмерный массив, у которого длина строк одинакова.

Кроме этого, в C# можно создавать специальный тип двухмерного массива, который называется невыровненным массивом. *Невыровненный массив* — это внешний массив, состоящий из внутренних массивов разной длины. Следовательно, невыровненный массив можно использовать для создания таблицы, содержащей строки различной длины.

При объявлении невыровненных массивов для указания каждой размерности используется отдельная пара квадратных скобок. Например, двухмерный невыровненный массив объявляется следующим образом:

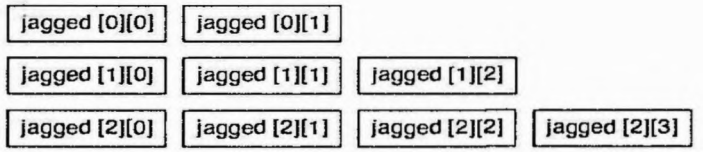
```
type[] [] array-name = new type[size] [] ;
```

Здесь слово `size` обозначает количество строк в массиве. Размерность строк не указывается, и память для них не выделяется. Для каждой строки необходимо выполнение отдельного оператора `new`. Такая технология позволяет определять строки различной длины. Например, в следующем фрагменте кода при объявлении массива `jagged` память выделяется только для внешнего массива, а потом при

выполнении трех операторов память выделяется отдельно для каждого внутреннего массива:

```
int[][] jagged = new int[3][];
jagged[0] = new int[2];
jagged[1] = new int[3];
jagqed[2] = new int[4];
```

После выполнения последовательности этих операторов невыровненный массив выглядит следующим образом:



Рассмотрев эту схему, вы поймете, почему невыровненный массив получил такое название.

Когда невыровненный массив создан, доступ к его элементам можно получить, указав каждый индекс в паре квадратных скобок. Например, чтобы присвоить значение 10 элементу массива `jagged[2][1]`, необходимо использовать следующий оператор:

```
lagged[2][1] = 10;
```

Вспомните, что для доступа к элементам прямоугольного массива применяется иная форма записи.

Ниже представлена программа, в которой используется невыровненный двухмерный массив. В массиве сохраняются данные о количестве пассажиров, пользующихся услугами маршрутного такси, чтобы добраться до аэропорта. Поскольку маршрутное такси совершает неодинаковое количество рейсов в различные дни недели (десять рейсов в рабочие дни и два по субботам и воскресеньям), для хранения данных можно использовать невыровненный массив `riders`. Заметьте, что вторая размерность для первых пяти строк имеет значение 10, а для оставшихся двух строк — 2.

```
// В программе демонстрируется использование невыровненного массива.
using System;

class Ragged {
  public static void Main() {
    int[][] riders = new int[7][];
    riders[0] = new int[10];
    riders[1] = new int[10];
    riders[2] = new int[10];
    riders[3] = new int[10];
    riders[4] = new int[10];

    riders[5] = new int[2];
    riders[6] = new int[2];

    int i, j;

    // Элементам массива присваиваются некоторые произвольные значения.
    for(i=0; i < 5; i++)
      for(j=0; j < 10; j++)
        riders[i][j] = i + j + 10;
```

Здесь внутренние массивы состоят из 10 элементов.

Здесь внутренние массивы состоят из 2 элементов.

```
    for(i=5; i < 7; i++)
      for (]=0; j < 2; j++)
        riders[i][j ] = i + j + 10;

    Console.WriteLine("Количество пассажиров, перевезенных за один рейс"+
                      "\nв будние дни: ");
    for(i=0; i < 5; i++) {
      for(j=0; j < 10; j++)
        Console.Write(riders[i][j) + " ");
      Console.WriteLine();
    }
    Console.WriteLine();

    Console.WriteLine("Количество пассажиров, перевезенных за один рейс"+
                      "\nв выходные дни: ");
    for(i=5; i < 7; i++)   {
      for(j=0;  j < 2; j++)
        Console.Write(riders[i][j] + " ");
      Console.WriteLine();
    }
  }
}
```

Невыровненные массивы редко используются в приложениях, но в некоторых ситуациях их применение может оказаться весьма эффективным. Например, если вам требуется очень большой двухмерный массив, который будет заполнен не полностью (то есть в нем используются не все элементы), то удобно использовать именно невыровненный массив.

Минутный практикум

1. Как указывается каждая размерность для прямоугольных многомерных массивов?
2. Может ли отличаться длина каждого внутреннего массива в невыровненном массиве?
3. Как инициализируются многомерные массивы?

Присваивание ссылок на массив

Когда значение одной переменной ссылочного типа, которая ссылается на массив, присваивается другой переменной такого типа, то второй переменной присваивается ссылка на тот же массив, на который ссылалась первая переменная. При этом не создается копия массива и не копируется содержимое одного массива в другой. В качестве примера рассмотрим следующую программу:

```
// Присваивание значения одной переменной ссылочного а-ипа, которая ссылается
// на массив, другой переменной такого же типа,
using System;
```

1. Размерности, разделенные запятыми, указываются в одних квадратных скобках.

2. Да.

3. Для инициализации многомерного массива список значений каждой размерности помещается в отдельный блок, заключенный в фигурные скобки. Вся эта конструкция указывается после оператора присваивания.

```
class AssignARef {
  public static void Main() {
    int i;

    int[] nums1 = new int[10];
    int[] nums2 = new int[10];

    for(i=0; i < 10; i++) nums1[i] = i;

    for(i=0; i < 10; i++) nums2[i] = -i;

    Console.Write("Значения элементов массива nums1: ");
    for(i=0; i < 10; i++)
      Console.Write(nums1[i] + " ");
    Console.WriteLine();

    Console.Write("Значения элементов массива nums2: ");
    for(i=0; i < 10; i++)
      Console.Write(nums2[i] + " ");
    Console.WriteLine();

    nums2 = nums1; // После выполнения этого оператора переменная nums2
                   // ссылается на тот же массив, что и переменная nums1.

    Console.Write("Значения массива, на который ссылается переменная nums2, "+
                  "\nпосле присваивания ей ссылки: ");
    for(i=0; i < 10; i++)
      Console.Write(nums2[i]    + " ");
    Console.WriteLine();

   // Теперь можно изменять значения массива, на который ссылается
   // переменная nums1, используя переменную nums2.
   nums2[3] = 99;

    Console.Write("Значения элементов массива nums1 после изменения," +
                  " выполненного с помощью \nпеременной nums2: ");
    for(i=0; i < 10; i++)
      Console.Write(numsl[i] + " ");
    Console.WriteLine();
  }
}
```

Результат выполнения этой программы следующий:

```
Значения элементов массива nums1: 0 1 2 3 4 5 6 7 8 9
Значения элементов массива nums2: 0 -1 -2 -3 -4 -5 -6 -7 -8 -9
Значения массива, на который ссылается переменная nums2
После присвоения ей ссылки: 0 1 2 3 4 5 6 7 8 9
Значения элементов массива numsl после изменения, выполненного с помощью
переменной nums2: 0 1 2 99 4 5 6 7 8 9
```

Как видите, после присваивания значения переменной `nums1` переменной `nums2` обе переменные массива ссылаются на один и тот же объект.

Использование свойства `Length`

В C# массивы реализованы как объекты, и это дает определенные преимущества при работе с массивами. Одно из них — возможность использования свойства `Length`.

Каждый массив имеет ассоциированное с ним свойство Length, в котором содержится информация о максимальном количестве элементов массива (то есть в массиве (имеется в виду класс) определено поле, содержащее информацию о длине массива). Ниже представлена программа, в которой используется свойство Length.

```
// В программе демонстрируется использование свойства Length.
using System;

class LengthDemo {
  public static void Main() {
    int[] list = new int[10];
    int[] nums = { 1, 2, 3);

    int[][] table = new int[3][]; // Невыровненный массив table.
    // Выделение памяти для внутренних массивов,
    table[0] = new int[] {1, 2, 3};
    table[1] = new int[] {4, 5};
    tabie[2] = new int[] {6, 7, 8, 9};

    Console.WriteLine("Длина массива list = " + list.Length);
    Console.WriteLine("Длина массива nums = " + nums.Length);
    Console.WriteLine("Длина массива table = " + table.Length);
    Console.WriteLine("Длина массива table[0] = " + table[0].Length);
    Console.WriteLine("Длина массива table[l] = " + table[1].Length);
    Console.WriteLine("Длина массива table[2] = " + table[2].Length);
    Console.WriteLine();

    // Использование свойства Length для инициализации массива list.
    for(int i=0; i < list.Length; i++)
      list[i] - i * i;

    Console.Write("Значения элементов массива list: ");
    // Использование свойства Length для отображения значений элементов
    // массива list.
    for(int i=0; i < list.Length; i++)
      Console.Write(list[i] + " ");
    Console.WriteLine();
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Длина массива list = 10
Длина массива nums = 3
Длина массива table = 3
Длина массива table[0] = 3
Длина массива table[1] = 2
Длина массива table[2] = 4

Значения элементов массива list: 0 1 4 9 16 25 36 49 64 81
```

Особое внимание следует обратить на использование свойства Length в двухмерном невыровненном массиве table. Мы уже говорили, что двухмерный невыровненный массив — это массив массивов. Следовательно, при использовании выражения

```
table.Length
```

мы получим количество массивов, которые составляют массив table (в данном случае это количество равно 3). Для получения значения длины каждого внутреннего массива во внешнем массиве table необходимо использовать следующее выражение:

```
table[0].Length
```

В данном случае мы получим значение длины внутреннего массива, имеющего индекс 0.

Обратите внимание на то, что в классе LengthDemo в условном выражении цикла for используется значение свойства list.Length. Поскольку каждый массив хранит информацию о своей длине, ее можно использовать в выражениях. Это позволяет создавать код, который может работать с массивами различной длины. Значение свойства Length не имеет ничего общего с количеством элементов, фактически хранящихся в массиве. Поле Length содержит информацию о числе элементов, которые может хранить массив.

Использование свойства Length упрощает работу многих алгоритмов, делая более эффективными некоторые операции с массивами. Например, в следующей программе при копировании значений одного массива в другой свойство Length используется для предотвращения возможности выхода за граничные значения индексов массива (что может привести к исключительной ситуации во время выполнения программы):

```
// В программе демонстрируется использование свойства Length для копирования
// одного массива в другой,
using System;

class ACopy {
  public static void Main() {
    int i;
    int[] nums1 = new int[10];
    int[] nums2 = new int[10];

    for(1=0; i < nums1.Length; i++) nums1[i] = i;

    // Копирование массива nums1 в массив nums2.

    if(nums2.Length >— nums1.Length)
      ford = 0; i < nums2.Length; i++)
        nums2[i] = nums1[i];

    for(i=0; i < nums2.Length; i++)
      Console.Write(nums2[i] + " ");

  }
}
```

Здесь свойство Length является ключевой составляющей выполнения двух функций. Во-первых, значение свойства Length используется для выполнения проверки того, позволяет ли размер массива nums2 скопировать в него все элементы массива nums1. Во-вторых, свойство Length используется в условном выражении цикла for, с помощью которого осуществляется копирование. Конечно, это очень простой пример, но такая технология может применяться и в гораздо более сложных алгоритмах.

Минутный практикум

1. Справедливо ли утверждение, что при присваивании значения одной переменной ссылочного типа, которая ссылается на массив, другой такой же переменной элементы первого массива копируются во второй.
2. Что собой представляет свойство `Length`?

Проект 5-2. Класс Queue

`Qdemo.cs`

Как вы знаете, *структуры данных* отличаются по способу организации данных. Простейшей структурой данных является массив, представляющий собой линейный список. Для доступа к элементам массива указывается его имя и индекс элемента. Массивы часто используются в качестве основы при построении более сложных структур данных, таких как стеки (stacks) и очереди (queues). *Стек* — это список, доступ к элементам которого осуществляется только по принципу Fll.O (first in, last out — «первым вошел, последним вышел»). *Очередь* — это список, доступ к элементам которого осуществляется только по принципу FIFO (first in, first out — «первым вошел, первым вышел»). То есть стек можно сравнить со стопкой тарелок — самая нижняя тарелка будет использована последней, а очередь можно сравнить с очередью в банке — кто первым пришел, того первым обслужат.

Стеки и очереди удобны тем, что в них средства хранения информации объединены с методами, которые обеспечивают доступ к этой информации. Таким образом, стеки и очереди представляют собой *механизмы доступа к данным*, в которых хранение и извлечение данных обеспечиваются самой структурой данных без помощи программы. Такое объединение, безусловно, соответствует конструкции класса. В этом проекте будет создан простой класс `Queue`.

В целом очереди поддерживают два базовых метода — `put` (поместить) и `get` (извлечь). При выполнении каждого метода `put` элементу массива, находящемуся в конце очереди (то есть имеющему индекс, соответствующий текущему состоянию очереди), присваивается некоторое значение. При выполнении каждого метода `get` из элемента массива, находящегося в начале очереди, считывается значение. Значение элемента может быть считано только один раз. Очередь считается заполненной, если отсутствуют свободные элементы для хранения новых значений, и считается пустой, если все значения элементов очереди считаны.

Существуют два основных вида очереди — круговая и некруговая. В круговой очереди при считывании значений «освободившиеся» элементы используются повторно. (На самом деле элементы не «освобождаются» — просто в соответствии с алгоритмом работы круговой очереди им «позволено» присваивать новые значения.) В некруговой очереди этого не происходит, поэтому такая очередь в итоге исчерпывает номера индексов своих элементов. Для простоты в данном примере мы создаем нскруговую очередь, но, приложив небольшие усилия, вы сможете преобразовать ее в круговую.

---

1. Нет. Меняется только ссылка.
2. `Length` — это свойство, которое имеется в каждом массиве. Оно содержит информацию о количестве элементов, которые могут храниться в массиве.

Пошаговая инструкция

1. Создайте файл и назовите его `QDemo.cs`.
2. Существуют различные способы создания класса `Queue`, но в данном проекте в основе структуры очереди лежат массив и методы, предназначенные для работы с элементами массива. То есть значения, помещаемые в очередь, будут храниться и элементах массива. Доступ к элементам этого массива будет осуществляться с помощью двух индексов. Индекс `putloo` указывает, какому элементу массива будет присвоено следующее значение, а индекс `getloo` указывает, из какого элемента массива будет считано следующее значение. В соответствии с конструкцией некруговой очереди после считывания значения какого-либо элемента повторное считывание его значения не допускается. Создаваемая в данном случае очередь будет хранить символы, но такая же конструкция может использоваться для хранения объектов любого типа. Начните создание класса `Queue` со следующих строк кода:

```
class Queue {
  public char[] q; // Этот символьный массив является основой очереди.
  public int putloc, getloc; // Индексы, указывающие, какому элементу массива
                             // будет присваиваться значение и из какого
                             // элемента значение будет считываться.
```

3. Ниже представлен конструктор `Queue()`, который создает очередь заданной длины:

```
  public Queue(int size) {
    q = new char [size+1];   // Выделение памяти для очереди.
    putloc — getloc — 0;
  }
```

Обратите внимание, что длина создаваемой очереди на единицу больше, чем это указано в переменной `size`. Такое увеличение длины необходимо для реализации алгоритма некруговой очереди, согласно которому один элемент массива не будет использоваться. Следовательно, для соответствия указываемой длины очереди с се истинной «емкостью» размер массива должен быть на единицу больше. Индексам `putloc` и `geloc` изначально присваивается значение 0.

4. Добавьте в код класса метод `put()`, с помощью которого элементам массива присваиваются значения.

```
  // Помещает символ в очередь.
  public void put(char ch) {
    if(putloc==q.Length-1) {
      Console.WriteLine(" — Очередь заполнена.");
      return;
    }

    putloc++;
    q[putloc] = ch;
  }
```

Метод начинается с оценки условного выражения, в котором проверяется, заполнена ли очередь. Если значение переменной `putloc` равно максимальному индексу массива `q`, это означает, что в массиве больше нет элементов, которым можно было бы присвоить значение. Если же в массиве еще есть неиспользованные элементы, то значение переменной `putloc` увеличивается на единицу, а новое

значение, передаваемое методу put в качестве аргумента, присваивается элементу с индексом, равным значению переменной putloc. Следовательно, переменная putloc всегда является индексом последнего помещенного в массив элемента.

5. Для считывания значений элементов используется метод get(), представленный ниже.

```
  // Считывает значения элементов массива (очереди).
  public char get () {
    if (getloc == putloc) {
      Console.WriteLme(" - Очередь пустая.");
      return (char) 0;
    }

    getloc++;
    return q[getloc];
  }
}
```

Обратите внимание, что в самом начале с помощью условного выражения проверяется, помещено в очередь какое-либо значение или нет. Если переменные getloc и putloc имеют одинаковые значения (то есть являются индексами одного и того же элемента), то очередь является пустой. Поэтому конструктор Queue() присваивается значение 0 обеим этим переменным. Затем переменная getloc увеличивается на единицу, и с помощью оператора return значение соответствующего элемента возвращается подпрограмме, вызвавшей метод. Таким образом, значение переменной getloc всегда указывает позицию последнего извлеченного элемента.

6. Ниже представлен весь код программы QDemo.cs.

```
/*
   Проект 5-2.

   В программе демонстрируется использование класса Queue,
   предназначенного для работы с символами.
 */
using System;

class Queue {
  public char[] q; // Этот символьный массив является основой очереди,
  public int putloc, getloc; // Индексы, указывающие, какому элементу массива
                             // будет присваиваться значение и из какого
                             // элемента будет считываться значение.
  public Queue(int size) {
    q = new char[size+1];    // Выделение памяти для очереди,
    putloc = getloc = 0;
  }

  // Помещает символ в очередь.
  public void put(char ch) {
    if (putloc==q.Length-1) {
      Console.WriteLine(" - Очередь заполнена.");
      return;
    }

    putloc++;
    q[putloc] = ch;
  }
```

```
  // Считывает значения элементов массива (очереди).
  public char get() {
    if (getloc == putloc) (
      Console.WriteLine(" Очередь пустая.");
      return (char) 0;
    }

    getloc++;
    return q[getloc];
  }
}
// Демонстрируется использование класса Queue.
class QDemo {
  public static void Main() {
    Queue bigQ = new Queue(100);
    Queue smallQ = new Queue(4);
    char ch;
    int i;

    Console.WriteLine("Класс bigQ используется для хранения символов "+
                      " алфавита.");
    // Помещение 26 символов английского алфавита в очередь bigQ.
    for(i=0; i < 26; i++)
      bigQ.put((char) ('A' + i)};

    // Считывание и вывод на печать значений элементов очереди bigQ.
    Console.Write("Содержимое очереди bigQ: ");
    for(i=0; i < 26; i++)   {
      ch = bigQ.get();
      if(ch != (char) 0) Console.Write(ch);
    }

    Console.WriteLine("\n");


    Console.WriteLine("Тестирование очереди smallQ.");
    // В данном цикле выполняется попытка поместить значение
    // в заполненную очередь.
    for(i=0; i < 5; i++) {
      Console.Write("Попытка поместить в очередь символ " + (char) ('Z' = i));

      smallQ.put((char) ('Z' = i));

      Console.WriteLine();
    }
    Console.WriteLine( ;

    // В данном цикле выполняется попытка считать значение из пустой очереди.
    Console.Write("Содержимое очереди smallQ: ");
    for(i=0; i < 5; i++) {
      ch = smallQ.get():

      if(ch != (char) 0) Console.Write(ch);
    }
  }
}
```

7. В результате работы этой программы на экран будут выведены следующие строки:

```
Класс bigQ используется для хранения символов алфавита.
Содержимое очереди bigQ: ABCDEFGKIJKLMNOPQRSTUVWXYZ

Тестирование очереди smallQ.
Попытка поместись в очередь символ Z
Попытка поместить в очередь символ Y
Попытка поместить в очередь символ X
Попытка поместить в очередь символ W
Попытка поместить в очередь символ V Очередь заполнена.

Содержимое очереди smallQ: ZYXW  Очередь пустая.
```

8. Попробуйте самостоятельно модифицировать класс `Quene` таким образом, чтобы в нем использовались другие типы данных. Например, пусть значения, помещаемые в очередь, будут иметь тип `int` или `double`.

Цикл `foreach`

О том, что в C# определен цикл `foreach`, мы упоминали в главе 3, а сейчас расскажем о нем подробно.

Цикл `foreach` используется для обработки элементов *коллекции.* Коллекция — это группа объектов. В C# определены несколько типов коллекций, одним из которых является массив. Синтаксис цикла `foreach` выглядит так:

```
foreach(type var-name in collection) statement;
```

Здесь словосочетание `type var-name` — это тип и имя *итерационной переменной*, которая будет принимать значение элементов из коллекции при выполнении цикла `foreach`. Циклически обрабатываемая коллекция указана словом `collection`. В приводимых ниже программах используемой коллекцией будет массив. Следовательно, тип `type` должен быть таким же (или совместимым), как базовый тип массива. Обратите внимание, что при работе с массивом итерационная переменная доступна только для чтения. Таким образом, вы не можете изменить содержимое массива путем присваивания итерационной переменной нового значения.

Ниже представлена простая программа, в которой используется цикл `foreach`. Вначале создастся массив целочисленного типа, элементам которого присваиваются начальные значения. Затем эти значения выводятся на экран, и одновременно вычисляется их сумма.

```
// В программе демонстрируется использование цикла foreach.
using System;

class ForeachDemo {
  public static void Main() {
    int sum = 0;
    int[] nums = new int[10];

    // Элементам массива nums присваиваются некоторые начальные значения.
    for(int i = 0; i < 10; i++)
      nums[i] = i;

    // Использование цикла foreach для вывода на экран значений
```

```
    // всех элементон массива и подсчета их суммы.
    foreach(int x in nums) {
      Console.WriteLine("Значение элемента массива - " + x);
      sum += x;
    }
    Console.WriteLine("Сумма значений всех элементов массива = " + sum);
  }
}
```

Обработка массшза nums с помощью никла foceacii.

Результат выполнения этой программы выглядит следующим образом:

```
Значение элемента массива = 0
Значение элемента массива = 1
Значение элемента массива = 2
Значение элемента массива = 3
Значение элемента массива = 4
Значение элемента массива = 5
Значение элемента массива = 6
Значение элемента массива = 7
Значение элемента массива = 8
Значение элемента массива = 9
Сумма значений всех элементов массива = 45
```

Как видите, обработка элементов массива с помощью цикла `foreach` производится в порядке возрастания их индексов.

С помощью цикла `foreach` можно также обрабатывать многомерные массивы. При этом значения элементов возвращаются по строкам — от первого до последнего элемента в каждой строке.

```
// В программе демонстрируется использование цикла foreach для обработки
   // элементов двухмерного массива.
using System;

class ForeachDemo2 {
  public static void Main() {
    int sum = 0;
    int[,] nums = new int[3,5];

    // Элементам массива nums присваиваются некоторые значения.
    for (int i = 0; i < 3; i++)
      for (int j=0; j < 5; j++)
        nurns[i,j] - (i + 1) * (j+1);

    // Использование цикла foreach для вывода на экран значений
    // всех элементов массива и подсчета их суммы.
    foreach(int x in nums) {
      Console.WriteLine("Значение элемента массива - " + x);
      sum += x;
    }
    Console.WriteLine("Сумма значений всех элементов массива - " + sum);
  }
}
```

Результат выполнения этой программы аналогичен предыдущему.

```
Значение элемента массива = 1
Значение элемента массива = 2
Значение элемента массива = 3
Значение элемента массива = 4
Значение элемента массива = 5
```

```
Значение элемента массива = 2
Значение элемента массива = 4
Значение элемента массива = 6
Значение элемента массива = 8
Значение элемента массива = 10
Значение элемента массива = 3
Значение элемента массива = 6
Значение элемента массива = 9
Значение элемента массива = 12
Значение элемента массива = 15
Сумма значений всех элементов массива 90
```

Поскольку с помощью цикла `foreach` массив можно обрабатывать только от начала до конца, может сложиться впечатление, что возможности этого цикла ограничены. Но это не так. Многие алгоритмы требуют именно такого механизма обработки. Ниже представлен вариант класса `MinMax`, с которым мы уже работали в этой главе. Напомним, что он предназначен для нахождения минимального и максимального значений элементов массива.

```
/*  В программе демонстрируется использование цикла foreach для нахождения
    минимального и максимального значений элементов массива.
 */

using System;

class MinMax {
  public static void Main() {
    int[] nums = {   99, -10,  100123,  18, -978,
                    5623,  463,  -9,  287,  49};
    int min, max;

    min = max = nums[0];
    foreach(int val in nums) {
      if(val < min) min = val;
      if(val > max) max = val;
    }
    Console.WriteLine("Минимальное значение = " + min +
                      " Максимальное значение — " + max);
  }
}
```

В этой программе цикл `foreach` работает прекрасно, поскольку нахождение минимального и максимального значений требует проверки всех элементов. Кроме того, цикл `foreach` используется для вычислений среднего значения, поиска значения или элемента и копирования массива.

Минутный практикум

1. Какие функции выполняет цикл `foreach`?
2. Можно ли с помощью цикла `foreach` осуществлять доступ к элементам массива в обратном порядке — от наибольшего индекса к наименьшему?
3. Можно ли использовать цикл `foreach` для присваивания значений элементу массива?

---

1. С помощью цикла `foreach` осуществляется обработка элементов массива.
2. Нет.
3. Нет.

Строки

С точки зрения решения повседневных задач программирования одним из наиболее важных в C# является тип данных `string`. Во многих языках программирования строка — это массив символов, но в C# строки являются объектами. Следовательно, тип данных `string` в C# является ссылочным типом.

Объекты типа `string` использовались в программах с первой главы этой книги, потому что при создании строкового литерала фактически создавался объект типа `string`. Например, в следующем операторе

```
Console.WriteLine("В C# строки являются объектами.");
```

строка `в C# строки являются объектами.` автоматически становится объектом типа `string`. Следовательно, класс `string` в предыдущих программах использовался неявно. В этом разделе мы расскажем, как оперировать такими данными явно. Но поскольку класс `string` достаточно большой, здесь будут рассмотрены только его основные характеристики.

Создание объектов класса `String`

Самым простым способом создания объекта типа `string` является использование строкового литерала. Например, в представленном ниже операторе переменная `str` является переменной ссылочного типа `string`, которой присваивается ссылка на строковый литерал.

```
string str = "Это строковый литерал.";
```

В этом случае переменная `str` инициализируется символьной последовательностью

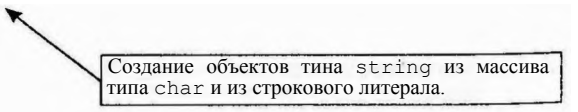
```
Это строковый литерал.
```

Вы также можете создать объект типа `string` из массива типа `char`. Например,

```
char[] str = {'t', 'e', 's', 't'};
string str = new string(str);
```

После создания объекта типа `string` его можно использовать в любом месте вашей программы, где разрешено применение строки в кавычках. Например, можно использовать объект типа `string` в качестве аргумента метода `WriteLine()`, как показано в следующей программе.

```
   // В программе демонстрируется создание объектов типа string.
using System;

 class StringDemo {
  public static void Main()     {

    char[] charray = {'Э', 'т', 'о',  ' ', 'с', 'т', 'р', 'о', 'к', 'а', '.'};
    string  str1  =  new  string(charray);
    string str2 = "Еще одна строка.";

    Console.WriteLine(str1);
    Console.WriteLine(str2);
  }
}
```



В результате работы данной программы будут выведены следующие строки:

```
Это строка.
Еще одна строка.
```

Операции со строками

Класс `string` содержит достаточно большое количество методов для работы со строками. Перечислим некоторые из них.

`static string Copy(string str)`	Возвращает копию переменной *str*
`int CompareTo(string str)`	Возвращает число меньше ноля, если значение вызывающей строки меньше значения переменной *str*; возвращает число больше ноля, если значение вызывающей строки больше значения переменной *str*; возвращает ноль, если строки равны
`int IndexOf(string str)`	Выполняет поиск в вызывающей строке подстроки, указанной в качестве параметра *str*. Возвращает индекс позиции в вызывающей строке, где первый раз была обнаружена подстрока, или значение -1, если подстрока не была обнаружена
`int LastIndexOf(string str)`	Выполняет поиск в вызывающей строке подстроки, указанной в качестве параметра *str*. Возвращает индекс позиции, где последний раз была обнаружена подстрока, или значение -1, если подстрока не была обнаружена

Объекты типа `string` имеют также свойство `Length`, содержащее информацию о длине строки.

Для получения значения отдельного символа строки необходимо использовать его индекс. Например,

```
string str = "test";
Console.WritrLine(str[0]);
```

В результате выполнения второго оператора будет выведен символ `t`. Как и в массивах, первый элемент строки имеет индекс 0. Но использовать индекс для присвоения символу в пределах строки нового значения нельзя. Индекс может использоваться только для получения символа.

Для сравнения двух строк можно использовать оператор ==. Обычно, когда этот оператор применяется к ссылкам на объект, он определяет, на один ли объект они ссылаются. С объектами типа `string` дело обстоит иначе. Когда оператор == применяется к двум ссылочным переменным типа `string`, то проверяется содержимое самих строк. То же самое справедливо для оператора `!=`; когда он используется при сравнении объектов типа `string`, то сравнивается содержимое самих строк. Однако другие операторы сравнения (например, < или >=) сравнивают ссылки, как и при работе с объектами иных типов.

Ниже представлена программа, в которой демонстрируется использование нескольких операций со строками.

```
// В программе демонстрируется использование некоторых операций со строками.
using System;
```

```
class StrOps {
  public static void Main() {
    string str1 = "Применение некоторых операций со строками.";
    string str2 = string.Copy(str1);
    string scr3 = "Это строковый литерал.";
    int result, idx;

    Console.WriteLine("Длина строки str1 - " +str1.Length + " символа.");

    // С помощью цикла for на экран вызолится строка str1
    // по одному символу за проход цикла.
    for(int i=0; i < str1.Length; i++)
      Console.Write(str1[i]);
    Console.WriteLine();

    if(str1 == str2)
      Console.WriteLine("str1 == str2");
    else
      Console.WriteLine("strl != str2");

    if (str1 == str3)
      Console.WriteLine("str1 == str3");
    else
      Console.WriteLine("str1 != str3");

    result = str1.CompareTo(str3);
    if (result == 0)
      Console.WriteLine("Строки str1 и str3 равны.");
    else if(result < 0)
      Console.WriteLine("Строка str1 меньше, чем str3.");
    else
      Console.WriteLine("Строка str1 больше, чем str3.");

    // Присвоение нового строкового литерала переменной str2.
    str2 = "Один Два Три Один.";
    Console.WriteLine("Один Два Три Один.");

    idx = str2.IndexOf("Один");
    Console.WriteLine("Индекс первого вхождения подстроки Один " +
                      "в строку str2: ", idx);
    idx = str2.LastlndexOf("Один");
    Consoie.WriteLine("Индекс последнею вхождения подстроки Один " +
                      "в строку str2: " + idx);
    }
  }
}
```

Результат выполнения программы следующий:

```
Длина строки str1 = 42 символа.
Применение некоторых операций со строками.
       str1 == str2
   str1 != str3
 Строка str1 меньше, чем str3.
 Один Два Три Один.
Индекс первого вхождения подстроки Один в строку str2: 0
Индекс последнего вхождения подстроки Один в строку str2: 13
```

С помощью оператора + вы можете выполнить *конкатенацию строк* (состыковать их вместе). Так, при выполнении блока операторов

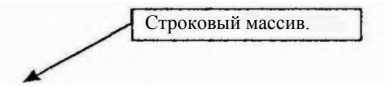
```
string str1 = "Конка"
string str2 = "тен";
string str3 = "ация";
string str4 = str1+str2+str3;
```

переменной `str4` присваивается строковый литерал `Конкатенация`.

Массивы строк

Как и другие типы данных, строки могут быть значениями элементов массива. Например, в следующей программе значениями элементов массива `str` являются четыре строковых литерала.

```
// В программе демонстрируется использование массива, значениями элементов
// которого являются строковые литералы.
using System;

 class StringArrays {
  public static void Main() {
     string[] str = {"Это", "пример", "строкового", "массива."};

     Console.WriteLine("Первоначальный массив: ");
     for(int i=0; i < str.Length; i++)
       Console.Write(str[i] + " ");
     Console.WriteLine("\n");

     // Изменение значений строкового массива.
     str[1] = "тоже";
     str[2] = "строковый";
     str[3] = "массив.";

     Console.WriteLine("Измененный массив: ");
     for (int i=0; i < str.Length; i++)
       Console.Write(str[i] + " ");
   }
}
```



Результат выполнения программы выглядит следующим образом:

```
Первоначальный массив:
Это пример строкового массива.

Измененный массив:
Это тоже строковый массив.
```

Неизменность строк

При работе со строками следует обратить внимание на очень важную характеристику объектов типа `string` — однажды созданная последовательность символов в строке не может быть изменена. Хотя такое ограничение на первый взгляд кажется серьезным недостатком, на самом деле это не так. Когда вам потребуется строка, отличная от уже существующей, то вы просто создадите новую строку, содержащую необходимые изменения. Поскольку неиспользуемые строковые объекты автоматически удаляются из памяти во время «сборки мусора», старые строки будут незаметно для вас удалены.

Значение переменной ссылочного типа, которая ссылается на объект типа `string`, безусловно, можно изменить. Но после создания нового объекта типа `string` его содержимое изменить уже нельзя.

Для изменения строки используется метод `Substring()`, возвращающий новую строку, которая содержит копию указываемой части вызывающей строки (строки, которая указывается слева от оператора точка (`.`)). Поскольку создается новый объект типа `string`, содержащий подстроку, то первоначальная строка остается неизменной. Далее мы будем использовать следующий синтаксис метода `Substring()`:

```
string Substring(int startindex, int len)
```

Здесь слово `startindex` — индекс элемента, с которого метод будет начинать копирование значений (символов). А с помощью слова `len` указывается длина копируемой подстроки. Ниже представлена программа, в которой демонстрируется использование метода `Substring()`.

```
// В программе демонстрируется использование метода Substring().
 using System;

class SubStr {
  public static void Main() {
    string orgstr = "Регистрация";

    // construct a substring
    string substr1 = orgstr.Substring(0, 7);
    string substr2 = orgstr.Substring(6, 5);

    Console.WriteLine("Строковый литерал orgstr: " + orgstr);
    Console.WriteLine("Строковый литерал substr1: " + substr1);
    Console.WriteLine("Строковый литерал substr2: " + substr2);

  }
}
```

В этом операторе создается новая строка, содержащая необходимую подстроку.

Результат выполнения этой программы следующий:

```
Строковый литерал orgstr: Регистрация
Строковый литерал substr1: Регистр
Строковый литерал substr2: рация
```

Первоначальная строка `orgstr` остается неизменной, а переменные `substr1` и `substr2` содержат подстроки.

Минутный практикум

1. Справедливо ли утверждение, что в C# все строки являются объектами?
2. Каким образом можно получить значение длины строки?
3. Какие функции выполняет метод `Substring()`?

---

1. Да.
2. Значение длины строки можно получить с помощью свойства `Length`.
3. Метод `Substring()` конструирует (возвращает) новую строку, являющуюся подстрокой вызывающей строки.

Ответы профессионала

Вопрос. Вы сказали, что после создания объекты типа `string` остаются неизменными. Понятно, что с практической точки зрения это не слишком серьезное ограничение. Но все-таки, существует ли возможность создавать строку, которую *можно* изменять?

Ответ. Это возможно. В C# предусмотрен класс `StringBuilder`, находящийся в пространстве имен `System.Text`. С помощью этого класса создаются строковые объекты, которые можно изменять. Однако для решения большинства задач вам, понадобятся объекты класса `string`, а не `StringBuilder`.

Побитовые операторы

В главе 2 были рассмотрены арифметические операторы, операторы сравнения и логические операторы C#. Эти операторы используются чаще всего, но в C# предусмотрены и дополнительные операторы, расширяющие возможности этого языка — побитовые операторы. Побитовые операторы работают непосредственно с битами своих операндов. Операндами побитовых операторов могут быть только целочисленные значения. Эти операторы не могут использоваться для значений типа `bool`, `float` или `double` либо для объектов каких-нибудь классов.

Такие операторы называются *побитовыми*, поскольку используются для проверки, установки или сдвига битов, которые составляют целочисленное значение. Побитовые операции важны для решения многих задач программирования на системном уровне (например, когда необходимо получение информации о состоянии устройства). Побитовые операторы перечислены в табл. 5.1.

Побитовые операторы AND, OR, XOR и NOT

Побитовые операторы `AND`, `OR`, `XOR` и `NOT` обозначаются соответственно, `&`, `!`, `^` и `~`. Они выполняют те же операции, что и их булевы логические эквиваленты, описанные в главе 2. Отличие состоит в том, что побитовые операторы воздействуют на операнды на уровне битов. В приведенной ниже таблице представлены результаты выполнения каждой операции е использованием единиц и нолей.

p	q	p&q	p|q	p^q	~p
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

Побитовый оператор `AND` можно представить как средство для сброса битов. То есть если какой-либо бит в одном из операндов имеет значение 0, то соответствующий ему бит в результате всегда будет иметь значение 0. Например,

```
  11010011
  10101010
& --------
  10000010
```

Таблица 5.1. Побитовые операторы

Оператор	Описание
&	Побитовый оператор AND
^ |	Побитовый оператор OR
	Побитовый оператор XOR
»	Оператор сдвига вправо
<<	Оператор сдвига влево
~	Унарный оператор NOT

В следующей программе демонстрируется использование оператора *&*. Программа преобразует символы нижнего регистра в символы верхнего регистра е помощью переустановки шестого бита в 0 (сброс шестого бита). Как определено в стандартном наборе символов Unicode/ASCII, значение символов нижнего регистра больше соответствующих значений символов верхнего регистра на число 32. Следовательно, чтобы преобразовать символы нижнего регистра в символы верхнего регистра, достаточно сбросить шестой бит, как это показано в программе.

```
// Программа преобразует символы нижнего регистра в символы верхнего
// регистра.
using System;

class UpCase {
  public static void Main() {
    char ch;

    for(int i=0; i < 10; i++)    {
      ch = (char) ('a' + i);
      Console.Write(ch);

      // При выполнении этого оператора сбрасывается шестой бит.
      ch = (char) (ch & 65503);     // Теперь значением переменной ch является
                                    // символ верхнего регистра.

      Console.Write(ch + " ") ;
    }
  }
}
```

В результате выполнения этой программы на экран будут выведены следующие пары символов:

```
aA bB cC dD eE fF gG hH iI jJ
```

Число 65503, используемое в операторе `AND`, имеет следующее двоичное представление: 1111 1111 1101 1111. Следовательно, операция AND оставляет без изменений все биты значения переменной `ch`, кроме шестого, который устанавливается в 0.

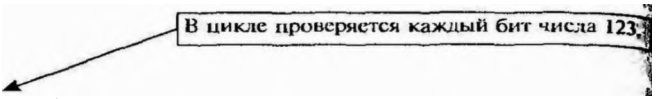
Операция AND также используется, когда необходимо определить, установлен или сброшен бит. Например, в приведенном ниже операторе производится проверка. Установлен ли четвертый бит в значении переменной `status`.

```
if(status & 8) Consol.WriteLine("Четвертый бит установлен.");
```

Значение 8 в этом операторе используется потому, что в двоичном представлении этого числа установлен только четвертый бит (0000 1000). Следовательно, оператор

if может иметь положительный результат только в том случае, если в значении переменной status также установлен четвертый бит. В следующей программе такой подход используется для вывода на экран двоичного представления значения типа byte.

```
// Программа выводит на экран двоичное представление числа, имеющего тип byte.
using System;

class ShowBits {
  public static void Main() {
    int t;
    byte val;

    val = 123;
    for(t=128; t > 0; t = t/2) {
      if((val & t) != 0) Console.Write("1 ");
      else Console.Write("0 ");
    }
  }
}
```



Результат выполнения этой программы следующий:

```
0 1 1 1 1 0 1 1
```

В цикле for с помощью побитового оператора AND проверяется каждый бит значения переменной val. Если бит установлен, то на экран выводится 1; если бит сброшен на экран выводится 0. В проекте 5-3 вы увидите, как можно расширить эту базовую концепцию опроса битов для создания класса, в котором бы определялись и выводились на экран биты любого значения целочисленного типа.

В противоположность оператору AND побитовый оператор OR может использоваться для установки битов. Если хотя бы в одном из операндов бит установлен, то соответствующий бит в значении переменной также будет установлен.

```
 11010011
 10101010
 --------
 11111011
```

Для преобразования символов верхнего регистра в символы нижнего регистра мы можем задействовать оператор OR, как это показано ниже:

```
// Программа преобразует символы верхнего регистра в символы нижнего регистра,
using System;

class LowCase {
  public static void Main() {
    char ch;

    for(int i=0; i < 10; i++) {
      ch = (char) ('A' + i);
      Console.Write(ch):

      // При выполнении этого оператора устанавливается шестой бит.
      ch = (char) (ch | 32);     // Теперь переменная ch содержит символ
                                 // нижнего регистра.
```

```
      Console.Write(ch + " ");
    }
  }
}
```

Ниже показан результат выполнения программы.

```
Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj
```

В программе операция OR применяется к каждому значению символа и значению 32, которое в двоичном представлении имеет вид 0000 0000 0010 0000. Следовательно, в двоичном представлении числа 32 установлен только шестой бит. Если операндами побитового оператора OR являются число 32 и любое другое число, то в результате мы получаем значение, в котором шестой бит установлен, а все остальные биты остаются неизмененными. В результате выполнения описанных выше действий в этой программе каждый символ верхнего регистра превращается в соответствующий символ нижнего регистра.

При выполнении операции XOR бит устанавливается только в том случае, если сравниваемые биты различны. Например,

```
  01111111
  10111001
^ --------
  11000110
```

Оператор XOR имеет очень интересное свойство, которое позволяет использовать его для шифровки сообщений. Если этот оператор применить к значениям X и Y, а затем к полученному результату и значению Y (вновь), то после выполнения второй операции будет получено значение x. То есть в последовательности операторов

```
R1 = X ^ Y;

R2 = R1 ^ Y;
```

элемент R2 имеет то же значение, что и x. Таким образом, результатом выполнения последовательности двух операций XOR с использованием одного значения будет первый операнд. Этот принцип можно использовать при создании простой программы для шифрования сообщений, в которой некоторое целочисленное значение будет являться ключом, используемым как для шифровки, так и для расшифровки сообщения с помощью оператора XOR. Для шифровки сообщения оператор XOR применяется первый раз, в результате мы получаем шифрованный текст. Для расшифровки оператор XOR применяется во второй раз, в результате мы получаем первоначальный текст. Ниже приведен код этой программы.

```
// В программе демонстрируется использование оператора XOR для шифровки и
// расшифровки сообщения.
using System;

class Encode {
  public static void Main() {
    string msg = "Совершенно секретно! Алекс Юстасу.";
    string encmsg = "";
    string decmsg = "";
    int key = 88;

    Console.Write("Первоначальное сообщение: ");
```

```
    Console.WriteLine(msg);

    // Шифрование сообщения.
    for(int i=0; i < msg.Length; i++)
      encmsg = encmsg + (char) (msg[i] ^ key);

    Console.Write("Зашифрованное сообщение: ");
    Console.WriteLine(encmsg);

    // Расшифровка сообщения.
    for(int i=0; i < msg.Length; i++)
      decmsg = decmsg + (char) (encmsg[i] ^ key);

    Console.Write("Расшифрованное сообщение: ");
    Console.WriteLine(decmsg);
  }
}
```

В результате выполнения этой программы на экран будет выведено как зашифрованное, так и расшифрованное сообщение:

```
Первоначальное сообщение: Совершенно секретно! Алекс Юстасу.
Зашифрованное сообщение:  ????ИА????хЙ??И?К??ухш???Йх?ЙК?ЙГГv
Расшифрованное сообщение: Совершенно секретно! Алекс Юстасу.
```

Как видите, в результате выполнения двух операций XOR с использованием одного и того же ключа мы получаем расшифрованное сообщение.

Унарный оператор NOT изменяет все биты операнда. Например, если некое целочисленное значение A имеет битовое представление 1001 0110, в результате выполнения операции ~A мы получим значение с битовым представлением 0110 1001.

В следующей программе демонстрируется работа оператора NOT. На экран выводится двоичное представление числа -34 и двоичное представление значения, полученного в результате применения к этому числу оператора NOT.

```
// В программе демонстрируется использование унарного оператора NOT.
using System;

class NotDemo {
  public static void Main() {
    sbyte b = -34;

    for(int t=128; t > 0; t = t/2) {
      if((b & t) != 0) Console.Write("1 ");
      else Console.Write("0 ");
    }
    Console.WriteLine();

    // Изменение значений всех битов числа на противоположные.
    b = (sbyte) ~b;

    for(int t=128; t > 0; t = t/2) {
      if ((b & t) != 0) Console.Write("1 ");
      else Console.Write("0 ");
    }
  }
}
```

результат выполнения этой программы имеет вид:

```
11011110
00100001
```

Операторы сдвига

В C# существует возможность сдвигать биты, составляющие значение, влево или вправо на заданное число позиций. В C# определены два оператора сдвига битов:

```
<< Сдвиг влево
>> Сдвиг вправо
```

Общий синтаксис этих операторов следующий:

```
value << num-bits
value >> num-bits
```

Здесь слово `value` — значение, в котором биты сдвигаются на определенное число позиций, указанное вместо словосочетания `num-bits`.

Сдвиг влево приводит к перемещению всех битов указанного значения влево на одну позицию, при этом освободившиеся младшие биты заполняются нолями, а соответствующее количество старших битов теряется. Сдвиг вправо приводит к перемещению всех битов на одну позицию вправо, при этом, если вправо сдвигаются биты беззнакового значения, старшие биты заполняются нолями, а соответствующее количество младших битов теряется. В случае сдвига вправо битов знакового целочисленного значения знаковый бит сохраняется (старшие биты заполняются единицами). В представлении отрицательного числа старший бит всегда имеет значение 1.

В следующей программе демонстрируется результат выполнения операции сдвига битов как вправо, так и влево. В первой части программы переменной типа `int` задается начальное значение 1, это означает, что установлен только его младший бит. Далее к этому значению применяется восемь операций сдвига, после выполнения которых на экран выводятся младшие восемь битов. Во второй части программы этой же переменной присваивается значение 128, к которому тоже применяется серия из восьми сдвигов, но уже вправо, с выводом результатов выполнения на экран.

```
// В программе демонстрируется использование операторов сдвига << и >>.
using System;

class ShiftDemo {
  public static void Main() {
     int val = 1;

     for(int i = 0; i < 8; i++) {
       for(int t-128; t > 0; t = t/2) {
         if ( (val & t) != 0) Console.Write("1 ");
         else Console.Write("0 ");
       }
       Console.WriteLine();
       val = val << 1; // Сдвиг влево.
     }
     Console.WriteLine();

     val = 128;
     for(int 1=0; i < 8; i++) {
```

```
      for (int t=128; t > 0; t = t/2) {
        if({val & t) != 0) Console.Wrxte("1 ");
        else Console.Write{"0 ");
      }
      Console.WriteLine();
      val = val >> 1; // Сдвиг вправо.
    }
  }
}
```

Ниже показан результат, демонстрирующий порядок выполнения операций сдвига

```
00000001
00000010
00000100
00001000
00010000
00100000
01000000
10000000

10000000
01000000
00100000
00010000
00001000
00000100
00000010
00000001
```

Ответы профессионала

Вопрос. Двоичное представление данных основано на степени числа 2. Могут ли операторы сдвига применяться для умножения или деления целочисленных значений на 2?

Ответ. Да. Побитовые операторы могут использоваться для выполнения очень быстрого умножения или деления на 2. Сдвиг влево удваивает значение. Сдвиг вправо уменьшает значение в два раза. Учтите, что результат будет верным, если при выполнении операции сдвига биты не сдвигаются за пределы двоичного представления этого числа.

Составные побитовые операторы

Все двоичные побитовые операторы могут использоваться одновременно с оператором присваивания. Например, в обоих операторах, представленных ниже, переменной `x` присваивается результат операции XOR, операндами в которой выступают значение переменной `x` и число 127.

```
x = X ^ 127;
x ^= 127;
```

Проект 5-3. Класс ShowBits

ShowBitsDemo.cs

В этом проекте создается класс ShowBits, позволяющий выводить на экран двоичное представление любого целочисленного значения. Такой класс может успешно использоваться в программировании. Например, если вы занимаетесь отладкой кода драйвера устройства, то, используя этот класс, сможете наблюдать поток данных в двоичном представлении.

Пошаговая инструкция

1. Создайте файл и назовите его ShowBitsDemo.cs.
2. Начните создание класса ShowBits с определения конструктора этого класса:

```
class ShowBits {
  public int numbits;

  public ShowBits (int n) {
    numbits = n;
  }
```

В классе ShowBits создаются объекты, предназначенные для отображения указанного количества битов. Например, для создания объекта, который будет выводить младшие восемь битов какого-либо целочисленного значения, необходимо использовать следующий оператор:

```
ShowBits b = new ShowBits(8);
```

Число битов, которые должны быть отображены, указывается конструктору и качестве аргумента. Его значение присваивается переменной numbits.

3. Для вывода битового представления числа в классе ShowBits определен метод show(), код которого показан ниже.

```
  public void show(ulong val) {
    ulong mask = 1;

    // Сдвиг маски (числа 1) в нужную позицию.
    mask <<= numbits-1;

    int spacer = 0;
    for(; mask != 0; mask >>= 1) {
      if((val & mask) != 0) Console.Write("1");
      else Console.Write("0");
      spacer++;
      if ((spacer % 8) ==  0) {
        Console.Write(" ");
        spacer = 0;
      }
    }
    Console.WriteLine ();
  }
}
```

Обратите внимание, что метод show() имеет один параметр типа ulong. Но это не означает, что методу show() всегда нужно передавать значение типа ulong. Поскольку в C# предусмотрено автоматическое приведение типов, методу show() может быть передано значение любого целочисленного типа. Количество выводимых битов

указывается значением переменной numbits. После каждых восьми битов метод show() вывешит пробел для облегчения чтения двоичных значений длинны битовых комбинаций.

4. Ниже представлен весь код программы ShowBitsDemo.

```
/*
   Проект 5—3.

   Класс, который предназначен для отображения двоичного представления
   целочисленного значения.
 */
using System;

class ShowBits {
  public int numbits;

  public ShowBits(int n) {
    numbits = n;
  }

  public void show(ulong val) {
    ulong mask = 1;

    // Сдвиг маски (числа 1) в нужную позицию.
    mask <<= numbits-1;

    int spacer = 0;
    for(; mask != 0; mask >>= 1) {
      if((val & mask) != 0} Console.Write("1");
      else Console.write("0");
      spacer++;
      if((spacer % 8) == 0) {
        Console.Write(" ");
        spacer = 0;
      }
    }
    Console.WriteLine();
  }
}

// Использование класса ShowBits.
class ShowBitsDemo {
  public static void Main() {
    ShowBits b = new ShowBits(8);
    ShowBits i = new ShowBits(32);
    ShowBits li = new ShowBits(64);

    Console.WriteLine("Двоичное представление числа 123: ");
    b.show(123);

    Console.WriteLine("\nДвоичное представление числа 87987: ");
    i.show(87987);

    Console.WriteLine("\nДвоичное представление числа 237658768: ");
    li.show(237658768);
```

```
    // С помощью объекта b класса ShowBits можно вывести на экран
    // младшие биты любого целочисленного значения.
    Console.WriteLine("\nМладшие 8 битов двоичного представления числа " +
                      " 87987: ");
    b.show(87987);
  }
```

5. Ниже показан результат выполнения программы `ShowBitsDemo`.

```
Двоичное представление числа 123:
01111011

Двоичное представление числа 87987:
00000000 00000001 01010111 10110011

Двоичное представление числа 237658768:
00000000  00000000 00000000 00000000 00001110 00101010 01100010 10010000

Младшие 8 битов двоичного представления числа 87987:
10110011
```

Минутный практикум

1. С какими типами данных могут работать побитовые операторы?
2. Какие функции выполняют операторы >> и <<?
3. Какая ошибка допущена в данном фрагменте кода?

```
byte val = 10;
val = val << 2;
```

Оператор ?

Одним из наиболее интересных операторов в C# является оператор `?`, который часто используется вместо цепочки операторов `if-else`, имеющей следующий синтаксис:

```
if (condition)
   var = expression!;
else
   var = expression2;
```

Здесь значение, присваиваемое переменной `var`, зависит от оценки условия (`condition`), которое управляет оператором `if`.

Оператор `?` называется *тернарным,* поскольку ему требуется три операнда. Синтаксис ею выглядит так:

```
Exp1 ? Exp2 : Exp3;
```

---

1. Побитовые операторы могут работать с целочисленными типами данных.
2. Оператор >> производит сдвиг битов вправо, а оператор << производит сдвиг битов влево.
3. При осуществлении операции сдвига влево значение переменной `val` преобразуется в тип `int`. Для обратного присваивания этого значения типа `int` переменной типа `byte` сначала необходимо выполнить приведение типа.

Здесь элемент Exp1 является булевым выражением, а элементы Exp2 и Exp3 — это выражения, типы которых должны быть одинаковыми. Обратите внимание на использование и расположение двоеточия.

Рассмотрим, как работает оператор ?. Вначале оценивается выражение Exp1. Если оно имеет значение true, то выполняется выражение Exp2, которое и становится результатом всего выражения ?. Если выражение Exp1 имеет значение false, то выполняется выражение Exp3, которое и становится результатом всего выражения ?.

Рассмотрим пример, в котором переменной absval присваивается абсолютное значение переменной val.

```
absval = val < 0 ? val; // Получение абсолютного значения переменной val.
```

В этом выражении переменной absval будет присвоено значение переменной val если значение больше или равно нолю. Если значение переменной val отрицательное, то переменной absval будет присвоено значение переменной val, взятое со знаком минус. (В результате выполнения оператора «унарный минус» отрицательное значение станет положительным.) Тот же код можно написать с использованием цепочки операторов if-else:

```
if(val < 0) absval = - val;
else absval = val;
```

В представленной ниже программе с помощью оператора ? выполняется делений двух чисел, но при этом запрещается деление на ноль.

```
// В программе демонстрируется использование тернарного оператора ? для
// предотвращения деления на ноль.
using System;

class NoZeroDiv {
  public static void Main() {
    int result;

    for(int i = -5; i < 6; i++) (
      result = i != 0 ? 100 / i : 0;
      if(i != 0)
        Console.WriteLine("100 / " + i + " = " + result);
    }
  }
}
```

Предотвращение деления на ноль.

Ниже представлен результат выполнения этой программы.

```
100 /-5 = -20
100 /-4 = -25
100 /-3 = -33
100 /-2 = -50
100 /-1 = -100
100 / 1 = 100
100 / 2 = 55
100 / 3 = 33
100 / 4 = 25
100 / 5 = 20
```

Особое внимание обратите на следующую строку кода:

```
result = i != 0 ? 100 / i : 0;
```

Здесь переменной result присваивается результат деления числа 100 на значение переменной i. Однако это деление производится только в том случае, если значение переменной i не равно нолю. В противном случае деление не выполняется, просто переменной result присваивается значение 0.

Фактически нет необходимости присваивать значение, получающееся в результате выполнения оператора ?, какой-либо переменной. Это значение можно использовать в качестве аргумента при вызове метода. Либо, если все выражения имеют тип bool, оператор ? можно использовать в качестве условного выражения в цикле или операторе if. Например, ниже представлена усовершенствованная версия предыдущей программы. Результаты выполнения обеих программ одинаковы.

```
// В программе демонстрируется использование тернарного оператора ? для
// предотвращения деления на ноль,
using System;

class NoZeroDiv2 {
  public static void Main() {

    for(int i = -5; i < 6; i++)
      if(i != 0 ? true : false)
        Console.WriteLine("100 / " + i +
                          " = " + 100 / i);
  }
}
```

Обратите внимание на использование оператора if. Если значение переменной i равно нолю, то результатом оценки условного выражения оператора if будет значение false. Таким образом предотвращается деление на ноль и не выполняется метод WruiteLine(). Во всех остальных случаях деление выполняется.

Контрольные вопросы

1. Покажите, как объявить одномерный массив, содержащий двенадцать элементов типа `double`.
2. Покажите, как объявить двухмерный массив 4x5 типа `int`.
3. Покажите, как объявить невыровненный двухмерный массив, и котором внешний массив состоит из пяти элементов.
4. Покажите, как инициализировать одномерный массив значениями от 1 до 5.
5. Объясните, как работает цикл `foreach`. Покажите его общий синтаксис.
6. Напишите программу, в которой для нахождения среднего значения из десяти чисел типа `double` применяется массив. Используйте любые десять чисел.
7. Измените сортировку в проекте 5-1 таким образом, чтобы производилась сортировка массива строк. Продемонстрируйте работу такой программы.
8. В чем состоит различие между методами классов `string indexOf ()` и `LastlndexOf()`?
9. Усовершенствуйте предназначенный для шифрования класс `Encode` таким образом, чтобы в качестве ключа использовалась 8-символьная строка.
10. Могут ли побитовые операторы работать сданными типа `double`?
11. Покажите, как можно переписать цепочку `if-else`

```
if(x < 0) y = 10;
else y = 20;
```

используя оператор `?`.

12. Является ли оператор `&` логическим оператором в следующем фрагменте кода? Почему?

```
bool a, b;
// ...
if(a & b); ...
```

Детальное рассмотрение методов и классов

- Управление доступом к членам класса
- Передача объектов методу
- Возвращение объектов методом
- Использование параметров ref и out
- Перегрузка методов
- Перегрузка конструкторов
- Возвращение значений методом Main()
- Передача аргументов методу Main()
- Рекурсия
- Ключевое слово static

В этой главе мы продолжим изучение классов и методов. Сначала рассмотрим вопросы управления доступом к членам класса, затем расскажем о передаче методам объектов и их возврате, перегрузке методов, далее проанализируем различные формы синтаксиса метода `Main()`, поговорим о рекурсии и функциях ключевого слова `static`.

Управление доступом к членам класса

Класс выполняет две основные функции, обеспечивающие инкапсуляцию данных. Во-первых, он связывает данные с кодом, который манипулирует ими. Преимущество такой связи в классе мы начали использовать еще в главе 4. Во-вторых, класс обеспечивает средства управления доступом к членам класса. Эту характеристику класса мы рассмотрим в данном разделе.

В C# доступ к членам класса осуществляется немного сложнее, чем в других языках программирования. В основном используются два базовых типа членов класса — открытые (`public`) и закрытые (`private`). Доступ к члену класса, объявленному как `public`, может свободно осуществляться кодом, определенным за пределами этого класса. До настоящего момента в примерах этой книги встречались члены класса с типом доступа (модификатором) `public`. Доступ к членам класса, объявленным как private, может осуществляться только другими методами этого же класса. Код, определенный за пределами класса, может получить доступ к закрытому члену класса, только используя открытые методы того же класса.

Ограничение доступа к членам класса является основным принципом объектно-ориентированного программирования, поскольку это ограничение защищает объекты от неправильного использования. Разрешая доступ к закрытым членам класса только посредством строго определенного набора методов, мы предотвращаем присвоение данным некорректных значений. Код, определенный за пределами класса, не может непосредственно присваивать значение закрытому члену класса. Способ и время использования данных в пределах объекта также можно строго контролировать. Следовательно, если класс правильно реализован, то он создает «черный ящик», который можно использовать, но вся его внутренняя работа защищена от несанкционированного вмешательства.

Модификаторы в C#

Управление доступом к членам класса осуществляется посредством четырех модификаторов: `public`, `private`, `protected` и `internal`. В этой главе мы рассмотрим только модификаторы `public` и `private`. Модификатор `protected` применяется лишь в тех случаях, когда задействовано наследование; подробно об этом модификаторе мы поговорим в главе 7. Модификатор `internal` используется в основном для компоновки программы (файла), проекта или компонента. Краткое описание модификатора `internal` будет дано в главе 12.

Если для члена класса используется модификатор `public`, то доступ к этому члену класса может осуществляться любым иным кодом программы, включая методы, определенные внутри других классов.

Если для члена класса указан модификатор `private`, то доступ к такому члену класса может осуществляться только другими членами этого же класса. Таким образом, у

методов одного класса отсутствует возможность доступа к закрытому члену другого класса. Как уже говорилось в главе 4, при отсутствии модификатора член класса автоматически становится закрытым в своем классе. Следовательно, при определении закрытых членов класса использование модификатора `private` не является обязательным.

В спецификации члена класса модификатор указывается первым (то есть объявление члена класса должно начинаться с модификатора). Приведем несколько примеров:

```
public  string  errMsg;
private double bal;
private bool isError(byte status) {   // ...
```

Чтобы вам легче было понять различие между модификаторами `public` и `private`, рассмотрим следующую программу:

```
// В программе демонстрируется использование модификаторов public и private.

using System;

class MyClass {
  private int alpha; // Модификатор private указан явно.
  int beta;          // Модификатор private назначается по умолчанию.
  public int gamma;  // Объявляется переменная с модификатором public.

  /* Определение методов, осуществляющих доступ к переменным alpha и beta.
     Члены класса (методы) имеют доступ к закрытым членам этого же класса
     (переменным).
   */
  public void setAlpha(int a) {
    alpha = a;
  }

  public int getAlpha()   {
    return alpha;
  }

  public void setBeta(int a) {
    beta = a;
  }

  public int getBeta() {
    return beta;
  }
}

class AccessDemo {
  public static void Main() {
    MyClass ob = new MyClass();

    /* Доступ к переменным alpha и beta возможен только с помощью методов
       класса MyClass.
     */
    ob.setAlpha(-99);
    ob.setBeta(19);
    Console.WriteLine("Значение переменной ob.alpha = " + ob.getAlpha());
    Console.WriteLine("Значение переменной ob.beta = " + ob.getBeta());
```

```
      // Два оператора, приводимые ниже, недействительны.
   // ob.alpha = 10;     // Неверно! Переменная alpha
                         // объявлена как private!
   // ob.beta = 9;       // Неверно! Переменная beta
                         // тоже закрытая!

   // А следующий оператор действительный, поскольку возможен непосредственней!
   // доступ к переменной, объявленной как public.
      ob.gamma = 99;
    }
}
```

Ошибка! Переменные alpha и beta являются закрытыми.

Верно, поскольку переменная gamma была объявлена с модификатором public.

Внутри класса `MyClass` переменная `alpha` объявлена как `private`, переменная `beta` имеет модификатор `private` по умолчанию, а переменная `gamma` объявлена как `public`. Поскольку переменные `alpha` и `beta` являются закрытыми, доступ к ним с помощью кода, определенного за пределами их класса, невозможен. Следовательно внутри класса `AccessDemo` ни к одной из двух указанных переменных нельзя обращаться непосредственно. Доступ к ним можно осуществлять только с использованием открытых методов, таких как `setAlpha()` и `getAlpha()`. Допустим, если удалить символы комментария в начале оператора

```
// ob.alpha = 10; // Неверно! Переменная alpha объявлена как private!
```

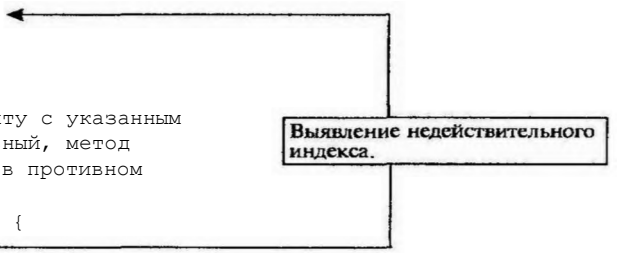
то эта программа не будет скомпилирована, поскольку в ней осуществляется попытка непосредственного обращения к закрытой переменной. Но, хотя доступ к переменной `alpha` не может осуществляться кодом, определенным за пределами класса `MyClass` эта переменная доступна для методов, определенных внутри класса `MyClass`, такая как `setAlpha()` и `getAlpha()`. То же самое справедливо по отношению к переменной `beta`.

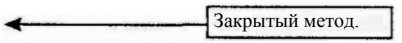
Чтобы продемонстрировать использование модификаторов доступа для решения практических задач, рассмотрим следующую программу, в которой создается массив типа `int`. В нем реализована защита от обращений к несуществующим элемента. массива, то есть обрабатываются ситуации, когда указывается индекс, выходящий за граничные значения индексов данного массива, — `< 0` или `> Length-1`. Алгоритм такой обработки мы в дальнейшем будем называть алгоритмом предотвращении сбоев, поскольку он предупреждает возникновение ошибки выполнения программы при которой система генерирует соответствующее сообщение, после чего выполнении программы прекращается. Реализация этого алгоритма осуществляется посредством инкапсуляции массива как закрытого члена класса, при этом доступ к массиву разрешен только методам, являющимся членами данного класса. При таком подход можно предотвратить любую попытку доступа к элементу массива по недействительному индексу. Далее представлен код класса `FailSoftArray`, в котором реализовал упомянутый выше алгоритм.

```
// В этом классе реализован алгоритм предотвращения сбоев.

using System;

class FailSoftArray {
  private int[] a; // Объявление ссылки на массив.
  private int errval; // Объявление переменной, которой будет присвоено
                      // значение, возвращаемое методом get() в случае
                      // выхода за граничные значения индексов массива.
```

Закрытые переменные экземпляра.

```
  public int Length; // Переменная Length объявлена как public.

  /* конструктор создает массив, получая в качестве параметров значение длины
     массива и значение, возвращаемое методом get() в случае
     выхода за граничные значения индексов массива.
  */
   public FailSoftArray(int size, int errv) {
    a = new int[size];
    errval = errv;
    Length = size;
  }

  // Метод возвращает значение элемента с индексом, переданным ему в
  // качестве параметра.
  public int get (int index) {
    if(os(index)) return a[index];
    return errval;
  }

  // Метод присваивает значение элементу с указанным
  // индексом. Если индекс действительный, метод
  // возвращает булево значение true, в противном
  // случае — значение false,
  public bool put (int index, int val) {
    if(ok(index))  {
      a[index] = val;
      return true;
    }
    return false;
  }

  // Метод возвращает значение true, если индекс не выходит за граничные
  // значения индексов массива.
  private bool ok(int index) {
   if(index >= 0 & index < Length) return true;
   return false;
  }
}

// Демонстрируется использование массива, в котором реализован алгоритм
// предотвращения сбоев.
class FSDemo {
  Public static void Main()    {
    FailSoftArray fs = new FailSoftArray {5, -1};
    int x;

    // Согласно определению метода put() при попытке обращения к
    // несуществующим элементам массива он просто возвращает значение false.
    Console.WriteLine("Использование алгоритма предотвращения сбоев.");
    for(int i=0; i < (fs.Length * 2); i++)
      fs.put(i, i*10);

    Console.Write("Значения элементов массива: ");
    for (int i=0; i < (fs.Length * 2); i++) {
      x = fs.get(i);
      if(x != -1) Console.Write(x + " ");
```





```
    }
    Console.WriteLine("")

    // Если при выполнении условною выражения в операторе if имеет место
    // попытка обращения к несуществующим элементам массива, то выводится
    // соответствующее сообщение.
    Console.WriteLine("\nОбработка ошибки с выводом сообщения.");
    for(int i=0; i < (fs.Length * 2); i++)
      if(!fs.put(i, i*10))
        Console.WriteLine("Индекс " + i + " выходит за граничные значения " +
                          "индексов данного массива.");

    Console.Write("Значения элементов массива: ");
    for (int i=0; i < (fs.Length * 2); i++) {
      x = fs.get(i);
      if (x != -1) Console. Write(x + " ");
      else
        Console.Write("\nИндекс " + i + " выходит за граничные значения " +
                      "индексов данного массива.");
    }
  }
}
```

Результат выполнения этой программы:

```
Использование алгоритма предотвращения сбоев.
Значения элементов массива: 0 10 20 30 40

Обработка ошибки с выводом сообщения.
Индекс 5  выходит за граничные  значения индексов данного массива.
Индекс 6  выходит за граничные  значения индексов данного массива.
Индекс 7  выходит за граничные  значения индексов данного массива.
Индекс 8  выходит за граничные  значения индексов данного массива.
Индекс 9  выходит за граничные  значения индексов данного массива.
Значения элементов массива: 0 10 20 30 40
Индекс 5  выходит за граничные  значения индексов данного массива.
Индекс 6  выходит за граничные  значения индексов данного массива.
Индекс 7  выходит за граничные  значения индексов данного массива.
Индекс 8  выходит за граничные  значения индексов данного массива.
Индекс 9  выходит за граничные  значения индексов данного массива.
```

Давайте внимательно разберем этот пример. Внутри класса `FailSoftArray` объявляются три закрытых члена класса. Первый член класса — переменная `a`, содержит ссылку на массив, в котором будут храниться данные. Второй член класса — переменная `errval`, хранит значение, которое будет возвращено при обращении метода `get()` к несуществующему элементу массива. Третьим членом класса является метод `ok()`, который определяет, не выходит ли индекс за граничные значения индексов данного массива. Все эти три члена класса могут использоваться только другими членами класса `FailSoftArray`. Остальные члены класса имеют модификатор `public`, то есть могут использоваться любым другим кодом программы, в которой применяется класс `FailSoftArray`.

При создании объекта типа `FailSoftArray` нужно указать размер массива и значение, которое будет возвращено, если метод `get()` обратится к несуществующему элементу массива. Возвращаемое значение должно явно отличаться от значений, сохраняемых в массиве. После создания и массив, на который ссылается переменная `a`, и возвращаемое значение переменной `errval` становятся недоступными (то

есть защищенными от неправильного использования) для пользователей объекта типа `FailSoftArray`. Так, пользователь не может непосредственно указать индекс элемента массива `a`, поскольку этот индекс может оказаться ошибочным. Доступ к объекту возможен только с использованием методов `get()` и `put()`.

В данном случае метод `ok()` объявлен как `private` не из практических соображений, а лишь для демонстрации работы модификатора. Этот метод без всякого ущерба для программы можно было определить как `public`, так как он не изменяет объект. Но поскольку его использование предусмотрено только внутри класса `FailSoftArray`, он может быть определен и как `private`.

Обратите внимание, что переменная экземпляра `Length` объявлена как `public`, что позволяет непосредственно обращаться к этой переменной для получения значения длины массива `FailSoftArray`.

Чтобы присвоить значение элементу массива `FailSoftArray`, необходимо вызвать метод `put()` и указать для него в качестве параметра индекс элемента. Чтобы извлечь из массива значения элемента, необходимо вызвать метод `get()` и указать индекс элемента. Если указанный индекс находится за пределами допустимых значений, то метод `put()` возвращает значение `false`, а метод `get()` — значение переменной `errval`.

Поскольку по умолчанию члены класса определяются как закрытые, нет необходимости явно использовать для них модификатор `private`. Поэтому далее в нашей книге при объявлении закрытых членов класса модификатор `private` использоваться не будет. Просто запомните, что если при объявлении члена класса модификатор опущен, то этот член класса будет закрытым.

Ответы профессионала

Вопрос. Рассмотренный выше массив, использующий алгоритм предотвращения сбоев, защищен от доступа к несуществующим элементам массива, но это достигнуто за счет изменения обычного синтаксиса индексации (обращения к элементам) массива. Существует ли более эффективный способ создания массива, реализующего подобный алгоритм?

Ответ. Да. Из следующей главы вы узнаете, что в C# имеется специальный тип члена класса, *индексатор*, который позволяет индексировать объект класса так же, как массив. Кроме того, в главе 7 рассматривается более совершенный способ создания переменной `Length` и работы с ней посредством превращения ее в *свойство*.

Минутный практикум

1. Назовите модификаторы доступа в C#.
2. Для чего используются модификаторы `private` и `public`?
3. Какой доступ к члену класса устанавливается по умолчанию, если при объявлении члена класса отсутствует модификатор доступа?

---

1. Модификаторами доступа в C# являются `private`, `public`, `protected` и `internal`.
2. Доступ к члену класса, объявленного как `private`, имеют только другие члены его класса. Член класса, объявленный с модификатором `public`, доступен и для кода, который определен вне данного класса.
3. По умолчанию член класса определяется как `private`.

Проект 6-1. Усовершенствованный класс Queue

Queue.cs

Модификатор `private` можно использовать для усовершенствования класса `Queue` разработанного в главе 5 (проект 5-2). В предыдущей версии все члены класса `Queue` являются открытыми, что приводит к определенным проблемам. Раз все члены класса объявлены как `public`, то код программы, использующей класс `Queue`, может получать доступ к массиву этого класса непосредственно и имеет возможность обращаться к элементам массива вне очереди. Поскольку основная задача очереди обеспечение доступа к элементам по принципу FIFO, то обращение вне очереди нежелательно. Также следует учитывать возможность злонамеренного изменения значений индексов `putloc` и `getloc`, что приведет к неправильной работе очереди. Все эти проблемы можно легко решить, сделав часть членов класса `Queue` закрытыми.

Пошаговая инструкция

1. Скопируйте код класса `Queue` из проекта 5-2 в новый файл с именем `Queue.cs`.
2. В классе `Queue` удалите модификаторы `public` из операторов объявления массива `q` и переменных, предназначенных для хранения индексов `putloc` и `getloc`, как показано ниже:

```
// Усовершенствованная версия класса Queue.
class Queue {
  // Теперь эти члены класса по умолчанию становятся закрытыми.
  char[] q;              // Этот символьный массив является основой очереди.
  int putloc, getloc; // Индексы, указывающие, какому элементу массива
                         // будет присваиваться значение и из какого
                         // элемента значение Будет считываться.

  public Queue(int size) {
    q = new char[size+1]; // Выделение памяти для очереди.
    putloc = getloc = 0;
  }

  // Помещает символ в очередь.
  public void put(char ch) {
    if(putloc==q.Length-1) {
      Console.WriteLine("- Очередь заполнена.");
      return;
    }

    putloc++;
    q[putloc] = ch;
  }

  // Считывает значения элементов массива (очереди).
  public char get() {
    if(getloc == putloc) {
      Console.WriteLine("- Очередь пустая.");
      return (char) 0;
    }

    getloc++;
    return q[getloc];
  }
}
```

3. Преобразование модификатора членов класса q, putloc и getloc из открытого в закрытый не влияет на программу, в которой используется класс Queue. В новой версии мы защищаем класс Queue от возможного неправильного использования. Так, приводимые ниже два оператора, в которых производится попытка непосредственного доступа к членам класса, являются недействительными:

```
Queue test = new Queue(10);

test.q[0] = 'X';    // Ошибка!
test.putloc = -100;    // Ошибка!
```

4. Теперь члены класса q, putloc и getloc закрытые, а класс Queue строго обеспечивает порядок доступа к элементам очереди по принципу FIFO.

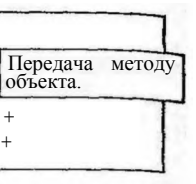
Передача объектов методу

До этого момента в программах нашей книги в качестве параметров методов использовались обычные типы значений, такие как int или double. Однако методу можно передавать и объекты. Рассмотрим простую программу, в которой сравниваются параллелепипеды, значения размеров которых хранятся в объектах класса Block.

```
// В программе демонстрируется передача методам объектов в качестве
// параметров.

using System;

class Block {
  int a, b, c;
  int volume;

  public Block(int i, int j, int k) {
    a = i;
    b = j;
    c = k;
    volume = a * b * c;
  }

  // Возвращает значение true, если в объекте ob содержатся
  // те же значенияразмеровпараллелепипеда,
  // что и в объекте, методкоторого был вызван.
  public bool sameBlock(Block ob) {
    if((ob.a == a) & (ob.b == b) & (ob.c == c)) return true;
    else return false;
  }

  // Возвращает значение true, если объем параллелепипеда
  // объекта ob такой же, как объем параллелепипеда
  // объекта, метод которого был вызван,
  public bool sameVoiume(Block ob) {
    if(ob.volume == volume) return true;
    else return false;
  }
}
```

Использование объекта в качестве параметра.

```
class PassOb {
  public static void Main() {
    Block ob1 = new Block(10, 2, 5);
    Block ob2 = new Block(10, 2, 5);
    Block ob3 = new Block(4, 5, 5);

    Console.WriteLine("Утверждение: объект ob1 имеет те же значения " +
                      "переменных, \nчто и о&ъект ob2 — " +
                       ob1.sameBlock(ob2));
    Console.WriteLine("Утверждение: объект ob1 имеет те же значения " +
                      "переменных, \nчто и объект ob3 — " +
                       ob1.sameBlock(ob2));
    Console.WriteLine ("Утверждение: объем параллелепипеда объекта ob1 " +
                      "\nравен объему параллелепипеда объекта ob3 — " +
                       ob1.sameVolume(ob3));
  }
}
```



Результат работы программы следующий:

```
Утверждение: объект ob1 имеет те же значения переменных,
что и объект ob2 — True
Утверждение: объект ob1 имеет те же значения переменных,
что и объект ob3 — False
Утверждение: объем параллелепипеда объекта ob1
равен объему параллелепипеда объекта ob3 - True
```

В методах `sameBlock()` и `sameVolume()` происходит сравнение значений переменных вызывающего объекта со значениями переменных объекта, переданного им в качестве аргумента. Метод `sameBlock()` сравнивает размеры параллелепипедов и возвращает значение `true` при полной их идентичности. Метод `sameVolume()` сравнивает объемы параллелепипедов, то есть результаты произведений значений всех трех переменных. В обоих случаях обратите внимание, что для параметра ob класс `Block` указан как его тип. Как видно из этого примера, синтаксически объекты передаются методам тем же способом, что и обычные типы данных.

Как передаются аргументы

Как показано в предыдущей программе, передача объектов методу является достаточно простой задачей. Однако в определенных ситуациях передача объекта может заметно отличаться от передачи обычных аргументов. Чтобы понять это, необходимо рассмотреть два способа передачи аргумента подпрограмме.

Первый способ называется *вызовом по значению*. При его использовании копию значения аргумента получает формальный параметр подпрограммы. Следовательно, изменение параметра подпрограммы не оказывает влияния на аргумент, который используется при ее вызове. Второй способ передачи аргумента называется *вызовом по ссылке*. В этом случае параметру передастся ссылка на аргумент (а не значение аргумента). Внутри подпрограммы эта ссылка используется для доступа к фактическому значению аргумента, который был указан при вызове метода. Это означает, что изменения параметра будут влиять на аргумент, используемый при вызове подпрограммы. В C# применяются оба эти способа передачи аргументов.

При передаче методу значений обычных типов, таких как `int` или `double`, используется способ передачи по значению. Следовательно, изменение параметра, получившего

аргумент, не влияет на аргумент за пределами метода. В качестве примера рассмотрим следующую программу:

```
// В программе аргументы простых типов передаются по значению.

 using System;

class Test {
   /* Выполнение этого метода не приведет к изменению переданного ему
      аргумента. */

  public void noChange(int i, int j) {
    i = i + j;
    j = -j;
   }
}

class CallByValue {
  public static void Main() {
    Test ob = new Test();

    int a = 15, b = 20;

    Console.WriteLine("Значения переменных a и b перед вызовом метода: " +
                      a + " и " + b);

    ob.noChange(a, b);

    Console.WriteLine("Значения переменных a и b после вызова метода:         " +
                      a + " и " + b) ;
  }
}
```

Результат выполнения данной программы будет следующим:

```
Значения переменных a и b перед вызовом метода: 15 и 20
Значения переменных a и b после вызова метода: 15 и 20
```

Как видите, операции, происходящие внутри метода `noChange()`, не влияют на значения переменных `a` и `b`, используемых в качестве аргументов при вызове метода.

При передаче методу ссылки на объект возникают некоторые трудности. Технически сама ссылка на объект передается по значению, следовательно, параметр получает копию ссылки, что по логике не должно изменять сам аргумент. (Например, передача параметру ссылки на новый объект не изменит объект, на который ссылается аргумент.) Но, и это весьма существенно, *изменение объекта,* на который ссылается параметр, приводит к изменению объекта, на который ссылается аргумент. Давайте Рассмотрим, почему так происходит.

Вспомните, что при создании переменной типа класса (ссылочного типа) вы создаете ссылку на объект, а не сам объект. Объект создается в памяти (ему выделяется память) с помощью оператора `new`, а ссылка на эту область памяти присваивается переменной ссылочного типа. При использовании переменной ссылочного типа в качестве аргумента для метода параметр принимает ссылку на тот же объект, на который ссылается аргумент. Следовательно, и аргумент, и параметр будут ссылаться на один и тот же объект. В действительности это означает, что объекты передаются методу с

использованием вызова по ссылке. Изменения объекта внутри метода *влияют* на объект, используемый в качестве аргумента. Рассмотрим следующую программу:

```
// В программе демонстрируется передача объектов по ссылке.

using System;

class Test {
  public int a, b;

  public Test(int i, int j) {
    a = i;
    b = j;
  }
  /* Передача методу объекта. Теперь значения переменных ob.a и ob.b в
     объекте, используемом при вызове метода, будут изменены.
   */
  public void change(Test ob) {
    ob.a = ob.a + ob.b;
    ob.b = -ob.b;
  }
}

class CallByRef {
  public static void Main() {
    Test ob = new Test(15, 20);

    Console.WriteLine("Значения переменных ob.a и ob.b перед вызовом метода: "
                + ob. a + " и " + ob. b);

    ob.change(ob);

    Console.WriteLine("Значения переменных ob.a и ob.b после вызова метода: "
                + ob.a + " и " + ob.b);
  }
}
```

В результате работы программы будет выведена следующая информация:

```
Значения переменных ob.a и ob.b перед вызовом метода: 15 и 20
Значения переменных ob.a и ob.b после вызова метода: 35 и -20
```

Как видите, при выполнении метода `change()` были изменены значения переменных объекта, а значит, и сам объект, используемый в качестве аргумента.

Итак, при передаче методу в качестве аргумента ссылки на объект сама ссылка передается с использованием вызова по значению, следовательно, создается копия этой ссылки. Но поскольку передаваемое значение ссылается на объект, копия этого значения ссылается на тот же объект, на который ссылается соответствующий аргумент.

Использование параметров с модификаторами ref и out

По умолчанию обычные типы значений, такие как `int` или `char`, передаются методу по значению. Это означает, что изменение параметра, который принимает обычный тип значений, не повлияет на аргумент, используемый в вызове. Но если применить

ключевые слова `ref` и `out`, то можно передать любое значение обычного типа по ссылке, что позволит методу изменить аргумент, используемый в вызове.

Прежде чем перейти к рассмотрению механизма использования ключевых слов `ref` и `out` необходимо выяснить, зачем может понадобиться передача значений обычных типов по ссылке, В этом возникает необходимость в двух случаях: во-первых, когда надо позволить методу изменять содержимое своих аргументов, во-вторых, когда надо позволить методу возвращать более одного значения. Давайте рассмотрим оба эти варианта подробно.

Очень часто возникает необходимость в использовании метода, способного изменять значения передаваемых аргументов. Наиболее типичным примером такого метода является метод `swap()`. Поскольку по умолчанию в C# обычные типы аргументов передаются по значению, нельзя создать метод, который переставит местами значения двух аргументов типа `int`. Для решения этой проблемы применяется модификатор `ref`.

Как вы знаете, оператор `return` позволяет методу возвращать значение вызывающей подпрограмме. Однако метод может возвращать *только одно* значение при каждом его вызове. Если же перед вами стоит задача возвратить две и больше единицы информации, то ее нельзя решить с помощью такого метода. К примеру, когда требуется создать метод для вычисления площади прямоугольника и определения, является ли этот прямоугольник квадратом, нужно, чтобы метод возвращал две единицы информации — площадь прямоугольника и значение, указывающее на равенство всех сторон. Для решения этой проблемы применяется модификатор `out`.

Использование модификатора `ref`

В C# использование модификатора параметра `ref` приводит к созданию вызова по ссылке вместо вызова по значению. Модификатор `ref` указывается при объявлении метода и при его вызове. Давайте рассмотрим простой пример. В следующей программе создается метод `sqr()`, возвращающий значение своего аргумента, возведенное во вторую степень, которое замещает начальное значение аргумента. Обратите внимание, что при объявлении метода модификатор `ref` предшествует типу параметра, а при вызове метода ключевое слово `ref` указывается непосредственно перед аргументом.

```
// В программе демонстрируется использование модификатора ref для передачи
// методу значения обычного типа по ссылке.

using System;

class RefTest {
  // Этот метод изменяет значение своего аргумента.
  public void sqr(ref int i) {
    i = i * i;
  }
}

class RefDemo {
  Public static void Main() {
    RefTest ob = new RefTest();

    int a = 10;
```

Здесь модификатор ref указывается в начале объявления параметра.

```
    Console.WriteLine("Значение переменной a перед вызовом метода: " + a);

    ob.sqr(ref a);

    Console.WriteLine("Значение переменной a после вызова метода: " + a);
  }
}
```

Здесь модификатор ref предшествует аргументу.

Результат выполнения этой программы, представленный ниже, подтверждает, что метод `sqr()` действительно модифицировал значение аргумента — переменной `a`.

```
Значение переменной a перед вызовом метода: 10
Значение переменной a после вызова метода: 100
```

Использование модификатора `ref` делает возможным создание метода, который будет менять местами значения своих аргументов (значения обычного типа). Ниже представлена программа, которая содержит метод `swap()`, меняющий местами значения двух целочисленных аргументов, передаваемых ему при вызове.

```
// Программа меняет местами значения двух переменных,

using System;

 class Swap {
  // Этот метод изменяет значения двух своих аргументов.
  public void swap(ref int a, ref int b) {
    int t;

    t = a;
    a = b;
    b = t;
  }
}

class SwapDeno {
  public static void Main() (
    Swap ob = new Swap();

    int x = 10, y = 20;

    Console.WriteLine("Значения переменных x и y перед вызовом метода: " +
                      x + " и " + y) ;

    ob.swap(ref x, ref y);

    Console.WriteLine("Значения переменных x и y после вызова метода:         " +
                      x + " и " + y) ;
  }
}
```

Результат выполнения этой программы:

```
Значения переменных x и y перед вызовом метода: 10 и 20
Значения переменных x и y после вызова метода: 20 и 10
```

Обратите внимание, что значение аргументу, передаваемому с модификатором `ref`, должно быть присвоено до вызова метода. Это важно, поскольку принимающий такой аргумент метод предполагает, что параметр ссылается на действительное значение. Следовательно, применяя модификатор `ref`, нельзя использовать метод для присваивания аргументу начального значения.

Использование модификатора `out`

Иногда параметр ссылочного типа используется для получения значения из метода, а не для передачи ему значения. Метод может выполнять некоторую функцию (например, открытие сокета) и возвращать в параметре ссылочного типа какое-либо значение, определенное для указания успешного или неудачного выполнения операции. Для этой цели в C# предусмотрен модификатор параметра `out`. При объявлении метода в круглых скобках указывается не информация, которую нужно передать методу, а информация, которая будет получена из метода. Необходимость использования еще одного модификатора, отличающегося от `ref`, объясняется тем. что параметр с модификатором `ref` должен быть инициализирован до вызова метода. Следовательно, только для выполнения данного условия необходимо присвоить аргументу некоторое (произвольное) значение.

Отличие между модификаторами `ref` и `out` заключается в том, что параметр с модификатором `out` может использоваться только для передачи значения из метода. При этом нет необходимости до вызова метода присваивать переменной начальное значение. Более того, внутри метода параметр с модификатором out всегда рассматривается как *не имеющий начального значения.* Метод *обязан* присвоить значение параметру, прежде чем передаст управление программой вызывающей подпрограмме. То есть после вызова метода параметр out всегда будет иметь значение.

В качестве примера рассмотрим программу, в методе которой используется параметр с модификатором `out`. Метод `rectInfo()` возвращает значение площади прямоугольника, для которого заданы размеры его сторон. В параметре `isSquare` метод возвращает значение `true`, если прямоугольник является квадратом, и значение `false` в противном случае. Таким образом, метод `rectInfo()` возвращает подпрограмме, которая его вызывает, две единицы информации.

```
// В программе демонстрируется использование модификатора параметра out.

using System;

class Rectangle {
  // Переменные содержат значения длины сторон прямоугольника.
  int side1;
  int side2;

  public Rectangle(int i, int j) {
    side1 = i;
    side2 = j;
  }


  // Метод возвращает значение площади прямоугольника и определяет, является
  // ли он квадратом.
  public int rectInfo(out bool isSquare) {
    if(side1==side2) isSquare = true;
    else isSquare = false;

    return side1 * side2;
  }
}
```

Передача информации из метода с использованием параметра с модификатором `out`.

```
class SwapDemo {
  public static void Main()   {
    Rectangle rect = new Rectangle(10, 23);
    int area;
    bool isSqr;

    area = rect.rectInfo(out isSqr);

    if(isSqr) Console.WriteLine("Данный прямоугольник является квадратом.");
    else Console.WriteLine("Данный прямоугольник не является квадратом.");

    Console.WriteLine("Площадь прямоугольника = " + area + ".");
  }
}
```

Обратите внимание, что до вызова метода `rectInfo()` параметру `isSqr` не присваивается начальное значение. Если бы параметр метода `rectInfo()` имел модификатор `ref`, это было бы невозможно. После завершения работы метода параметр `isSqr` содержит либо значение `true`, либо значение `false` в зависимости оттого, является прямоугольник квадратом или нет. Значение площали прямоугольника возвращается с помощью оператора `return`. Результат выполнения этой программы выглядит следующим образом:

```
Данный прямоугольник не является квадратом.
Площадь прямоугольника = 230.
```

Минутный практикум

1. Чем различается вызов по значению и вызов по ссылке?
2. Как в C# передаются значения обычных типов? Как передаются объекты?
3. Какие функции выполняет модификатор `ref`? Чем он отличается от модификатора `out`?

Ответы профессионала

Вопрос. Могут ли модификаторы `ref` и `out` применяться с параметрами ссылочного типа, например, при передаче ссылки на объект?

Ответ. Да. При использовании модификаторов `ref` и `out` для параметров ссылочного типа сама ссылка передастся по ссылке. Это позволяет метолу изменить направление ссылки, то сеть после выполнения этого метода переменная будет ссылаться уже на другой объект. Рассмотрим следующую программу:

```
// В программе демонстрируется использование модификатора ref при вызове
// по ссылке.
using System;

class Test {
  public int a;
```

1. При вызове по значению подпрограмме передается копия аргумента, а при вызове по ссылке — ссылка на аргумент.
2. В C# аргументы обычных типов передаются но значению, а объекты — по ссылке.
3. Модификатор `ref` создает вызов по ссылке для параметров обычных типов. Модификатор `out` также создает вызов по ссылке, но он не используется для передачи информации методу.

```
  public Test(int i) {
    a = i;
  }
  // При выполнении этого метода аргумент не будет изменен.
  public void noChange(Test o) {
    Test newob = new Test(0);
    o = newob; // Выполнение данного оператора не оказывает влияния на объект
                // "o" вне метода noChange().
  }

  // После выполнения этого метода переменная "o" будет ссылаться на другой
  // объект.
  public void change(ref Test o) {
    Test newob = new Test(0);
    o = newob; // Этот оператор изменит аргумент.
  }
}

class CallObjByRef {
  public static void Main() {
    Test ob = new Test(100);

    Console.WriteLine("Значение переменной ob.a после инициализации объекта: "
                      + ob.a);

    ob.noChange(ob);
    Console.WriteLine("Значение переменной ob.a после вызова метода " +
                        "noChange(): " + ob.a);

    ob.change(ref ob);
    Console.WriteLine("Значение переменной ob.a после вызова метода " +
                        "change():  " + ob.a);
  }
}
```

Ниже представлен результат выполнения этой программы.

```
Значение переменной ob.a после инициализации объекта: 100
Значение переменной ob.a после вызова метода noChange(): 100
Значение переменной ob.a после вызова метода change(): 0
```

В этой программе переменной `o` присваивается ссылка на новый объект внутри метода `noChange()`, что не влияет на объект `ob` внутри метода `Main()`. Однако внутри метода `change()`, в котором используется параметр с модификатором `ref`, присвоение переменной `o` ссылки на новый объект приводит к тому, что внутри метода `Main()` переменная `ob` будет ссылаться уже на другой объект.

Использование переменного количества аргументов

При создании метода обычно заранее известно количество аргументов, которые будут ему передаваться. Но это бывает не всегда, иногда требуется метод, которому можно передавать произвольное количество аргументов. В качестве такого примера рассмотрим метод, в котором производится поиск наименьших значений. Этому методу может быть передано два, три, четыре значения и т. д. С использованием обычных

параметров его создать нельзя. В данном случае необходимо задействовать параметр с модификатором `params`, с помощью которого можно передавать методу произвольное количество параметров.

Модификатор `params` используется для объявления в качестве параметра массива который будет способен принимать ноль и больше аргументов. Число элементов в массиве будет ранным числу аргументов, передаваемых метолу. Следовательно, для получения аргументов программа должна будет обращаться к массиву.

В приведенном ниже примере используется модификатор `params` для создание метода `minVal()`, который возвращает минимальное значение из нескольких, пере данных в качестве аргументов.

```
// В программе демонстрируется использование модификатора params.

using System;

class Min {
  public int minVai(params int[] nums) {
    int m;

    if(nums.Length == 0) {
      Console.WriteLine("Ошибка! Для метода не указаны аргументы.");
      return 0;
    }

    m — nums[0];
    for(int i=1; i < nums.Length; i++)
      if(nums[i] < m) m = nums[i];

    return m;
  }
}

class ParamsDemo {
  public static void Main() {
    Min ob = new Min();
    int min;
    int a = 10, b = 20;

    // Вызов метода с двумя аргументами.
    min = ob.minVal(a, b);
    Console.WriteLine("Минимальное значение = " + min);

    // Вызов метода с тремя аргументами.
    min = ob.minVal(a, b, -1);
    Console.WriteLine("Минимальное значение = " + min);

    // Вызов метода с пятью аргументами.
    min = ob.minVai(18, 23, 3, 14, 25);
    Console.WriteLine("Минимальное значение = " + min);

    // Методу в качестве аргумента также может быть передан целочисленный
    // массив.
    int[] args = { 45, 67, 34, 9, 112, 8 };
    min = ob.minVal(args);
    Console.WriteLine("Минимальное значение = " + min);
  }
 }
```

Создание параметра переменной длины с использованием модификатора `params`.

Программа выводит следующий результат:

```
Минимальное значение = 10
Минимальное значение = -1
Минимальное значение = 3
Минимальное значение = 8
```

Каждый раз при вызове метода `minVal()` ему передаются аргументы с помощью массива `nums`. Длина массива равна количеству элементов. Следовательно, метод `minVal()` можно использовать для поиска минимального значения из любого количества аргументов.

Хотя параметру с модификатором `params` можно передавать любое число аргументов, все они должны быть совместимы с типом массива, определенного и качестве параметра. Например, вызов метода `minVal()`

```
min >> ob.minVal (1, 2.2);
```

является неверным, поскольку значение `double(2.2)` не конвертируется автоматически в тип `int`, который имеет массив `nums` в методе `minval()`.

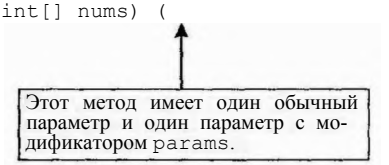
При использовании параметра с модификатором `params` необходимо контролировать число элементов массива, поскольку параметр с этим модификатором может принимать любое число аргументов (даже ноль). В частности, приведенные ниже операторы. в которых вызывается метод `minVal()`. являются синтаксически правильными.

```
min = ob.minVal(); // Вызов метода без аргументов.
min = ob.minVal(3);   // Вызов метода с одним аргументом.
```

Поэтому в методе `minVal()` перед попыткой обращения к первому элементу массива проверяется, есть ли в массиве хотя бы один элемент. В случае отсутствия таком проверки при вызове метода `minVal()` без аргументов возникнет исключительная ситуация, поскольку произойдет ошибка выполнения программы. (Далее при описании исключительных ситуаций мы расскажем о более эффективном способе защиты от такого рода ошибок.) Код метода `minVal()` написан таким образом, что разрешается производить его вызов с одним аргументом. При этом возвращается этот единственный аргумент.

Метод может иметь обычные параметры и параметр переменной длины. В следующем примере метод `showArgs()` принимает один параметр типа `string`, а затем параметр переменной длины (с модификатором `params`) — массив типа `int`:

```
// В программе демонстрируется использование обычного параметра и параметра
// переменной длины.

using System;

      class MyClass {
  public void showArgs(string msg, params int[] nums) (
    Console.Write(msg + ":");

    foreach(int i in nums)
      Console.Write(i + " ");

    Console.WriteLine();
  }
}

class ParamsDemo2 {
```



```
  public static void Main() {
    MyClass ob = new MyClass();

    ob.showArgs("В этом методе кроме строки есть еще 5 аргументов: ",
                1, 2, 3, 4, 5);

    ob.showArgs("А в этом метоле кроме строки есть еще 2 аргумента: ",
                17, 20);

  }
}
```

Результат выполнения программы следующий:

```
В этом методе кроме строки есть еще 5 аргументов: 1 2 3 4 5
А в этом методе кроме строки есть еще 2 аргумента: 17 20
```

В тех случаях, когда метод имеет обычные параметры и параметр с модификаторов `params`, параметр переменной длины должен в списке параметров располагаться последним. Кроме того, в любом случае у метода может быть только один параметр с модификатором `params`.

Минутный практикум

1. Как создать параметр, который принимает переменное количество аргументов?
2. Может ли метод иметь более одного параметра `params`?
3. Справедливо ли утверждение, что параметр `params` может располагаться в любом месте списка параметров?

Возвращение объектов

Метод может возвращать любые типы данных, включая объекты какого-либо класса. В следующей программе для сообщения об ошибке используется класс `ErrorMsg`, в котором определен строковый массив. Каждая строка представляет собой краткую характеристику ошибки. На основании кода ошибки, полученного в качестве аргумента, метод `getErrorMsg()` возвращает значение элемента массива — объект типа `string`.

```
// В программе демонстрируется возврат объектов.

using System;

class ErrorMsg {
  string [] msgs = {
    "Ошибка вывода.",
    "Ошибка ввода.",
    "Недостаточно свободного места на диске.",
    "Выход за граничные значения индексов массива."
```

1. Для создания параметра, который принимает переменное количество аргументов, используется модификатор `params`.

2. Нет.

3. Нет. Параметр `params` должен стоять последним в списке параметров.

```
  // Возврат сообщения об ошибке.
  public string getErrorMsg(int i) {
    if (i >=0 & i < msgs.Length)
      return msgs[i];
    else
      return "Введен неправильный код ошибки.";
  }
}

class ErrMsg {
  public static void Main()  {
    ErrorMsg err = new ErrorMsg();

    Console.WriteLine(err.getErrorMsg(2));
    Console.WriteLine(err.getErrorMsg(19));
  }
}
```

Возврат объекта типа `string`.

В результате работы этой программы будут выведены следующие строки:

```
Недостаточно свободного места на диске.
Введен неправильный код ошибки.
```

Таким же образом можно возвращать объекты создаваемых вами классов. Например, ниже представлена обновленная версия предыдущей программы, в которой создаются два класса, предназначенные для обработки ошибок. Один класс, инкапсулирующий сообщение об ошибке с кодом «серьезности» ошибки, называется `Err`. Второй класс содержит метод `getErrorInfo()`, который возвращает объект типа `Err`. Этот класс именуется `ErrorInfo`.

```
// В программе демонстрируется возврат объектов, определенных пользователем.

using System;

class Err {
  public string msg; // Сообщение об ошибке.
  public int severity; // Переменная, содержащая код серьезности ошибки.

  public Err(string m, int s) {
    msg = m;
    severity = s;
  }
}

class ErrorInfo {
    string[] msgs = {
    "Ошибка вывода.",
    "Ошибка ввода.",
    "Недостаточно свободного места на диске.",
    "Выход за граничные значения индексов массива."
  };
  int[] howbad = { 3, 3, 2, 4 };

  public Err getErrorInfo(int i) {
    if (i >=0 & i < msgs.Length)
      return new Err(msgs[i], howbad[i]);
    else
      return new Err ("Введен неправильный код ошибки. ", 0);
```

Возвращение объекта типа `Err`.

```
  }
}
class ErrInfo {
  public static void Main() {
    ErrorInfo err = new ErrorInfo();
    Err e;

    e = err.getErrorInfo(2);
    Console.WriteLine(e.msg + " Серьезность ошибки: " + e.severity);

    e = err.getErrorInfo(19);
    Console.WriteLine(e.msg + " Серьезность ошибки: " + e.severity);
  }
}
```

Эта программа выведет на экран следующую информацию:

```
Недостаточно свободного места на диске. Серьезность ошибки: 2
Введен неправильный код ошибки. Серьезность ошибки: 0
```

Каждый раз при вызове метода `getErrorInfo()` создастся новый объект типа `Err`, а ссылка на него возвращается вызывающей программе. Затем этот объект используется в методе `Main()` для вывода на экран сообщения об ошибке и кода «серьезности» ошибки. Объект, возвращенный методом, существует до тех пор. пока он упоминается в программе, его удаление будет выполнено только при «сборке мусора». Таким образом, объект не может быть уничтожен только потому, что создавший его метод возвратил управление программой вызывающей подпрограмме.

Перегрузка метода

Данный раздел познакомит вас с одним из самых замечательных свойств C# — перегрузкой методов. В этом языке два и более методов в пределах одного класса могут иметь одинаковые имена при условии, что объявления параметров в методах различаются. (Необходимо, чтобы в методах были указаны параметры различных типов, или методы должны различаться количеством параметров.) В этом случае методы называются *перегруженными,* а процесс определения метода с таким же именем называется *перегрузкой метода.* Перегрузка методов является одним из способов реализации полиморфизма в C#.

В общем случае для перегрузки метода вам достаточно объявить другую его версию, а компилятор позаботится обо всем остальном. Безусловно, перегружаемые методы могут отличаться и по типу возвращаемого значения, по этого недостаточно, чтобы два метода можно было считать различными. Типы возвращаемых значений не содержат достаточного количества информации для того, чтобы компилятор мог определить, какой из методов необходимо использовать. При вызове перегружаемого метода выполняется тот метод, параметры которого совпадают с аргументами.

Ниже представлена простая программа, в которой выполняется перегрузка метода.

```
// В программе демонстрируется перегрузка метода.

using System;

class Overload {
```

```
  public void ovlDemo()  {
    Console.WriteLine("Этот метод не имеет параметров.");
   }

   // Перегруженный метод ovlDemo, версия с одним параметром типа int.
  public void ovlDemo (int a) {
    Console.WriteLine("Метод с одним параметром: " + a);
   }

   // Перегруженный метод ovlDemo, версия с двумя параметрами типа int.
  public int ovlDemo (int a, int b) {
    Console.WriteLine("Метод с двумя параметрами: " + a + " и " + b);
    return a + b;
   }

   // перегруженный метод ovlDemo, версия с двумя параметрами типа double.
  public double ovlDemo(double a, double b) {
    Console.WriteLine("Метод с двумя параметрами типа double: " +
                      a + " и " + b);
    return a + b;

   }
 }
class OverloadDemo {
  public static void Main() {
    Overload ob = new Overload();
    int resI;
    double resD;

    // Вызов всех версий метода ovlDemo().
    ob.ovlDemo();
    Console.WriteLine();

    ob.ovlDemo(2);
    Console.WriteLine();

    resI = ob.ovlDemo(4, 6);
    Console.WriteLine("Результат выполнения метода ob.ovlDemo(4, 6): " +
                      resI);
    Console.WriteLine();


    resD = ob.ovlDemo (1.1, 2.32);
    Console.WriteLine("Результат выполнения метода ob.ovlDemo (1.1, 2.32): " +
                      resD);
  }
}
```

Первая версия метода.

Вторая версия метода.

Третьи версии метода.

Четвертая версия метода.

В результате выполнения всех этих методов программы будет выведена следующая информация:

```
Этот метод не имеет параметров.

Метод с одним параметром: 2

Метод с двумя параметрами: 4 и 6
Результат выполнения метода ob.ovlDemo(4, 6): 10
```

```
Метод с двумя параметрами типа double: 1,1 и 2,32
Результат выполнения метода ob.ovlDemo(1.1, 2.32): 3,42
```

В данной программе метод `oviDemo()` перегружен четыре раза. Первая версия не принимает ни одного параметра, вторая версия принимает один целочисленный параметр, третья версия принимает два целочисленных параметра, а четвертая версия принимает два параметра типа `double`. Обратите внимание, что первые две версии имеют возвращаемый тип `void`, а вторые две — возвращают значения. Такая программа является действительной. Но мы уже говорили, что перегруженные методы отличаются не только типом возвращаемого значения, поэтому попытка использования следующих двух версий приведет к ошибке:

```
// Если в программе присутствует только один метод ovlDemo(int),
// она будет работать.
public void ovlDemo(int a) {
    Console.WriteLine("Метод с одним параметром: " + a);
}

/* Ошибка! Два метода ovlDemo(int) должны отличаться
   не только типом возвращаемого значения.
/*
public int ovlDemo(int a) {
    Console.WriteLine("Метод с одним параметром: " + a);
    return a * a;
}
```

Типы возвращаемых значений не используются для определения различим между перегруженными методами.

В комментарии говорится, что различие в типе возвращаемых значений является недостаточным для перегрузки метода.

В главе 2 мы рассказывали, что в C# обеспечиваются автоматические преобразования определенных типов данных. Эти преобразования также применяются к параметрам перегруженных методов. Рассмотрим следующий пример:

```
// В программе демонстрируется автоматическое преобразование типов аргументов.

using System;

class Overload2 {
  public void f (int x) {
    Console.WriteLine("Значение переменной x внутри метода f(int): " + x);
  }

  public void f(double x) {
    Console.WriteLine("Значение переменной x внутри метода f(double): " + x);
  }
}

class TypeConv {
  public static void Main() {
    Overload2 ob = new Overload2();

    int i = 10;
    double d = 10.1;

    byte b = 99;
    short s = 10;
    float f = 11.5F;

    ob.f(i); // Вызывается метод ob.f(int)
```

```
    ob.f(d);  // Вызывается метод ob.f(double)

    ob.f(b);  // Вызывается метод ob.f(int) и выполняется автоматическое
              // преобразование типа аргумента.
    ob.f(s);  // Вызывается метод ob.f(int) и выполняется автоматическое
              // преобразование типа аргумента.
    ob.f(f);  // Вызывается метол ob.f(double) и выполняется автоматическое
              // преобразование типа аргумента.
  }
}
```

Ниже представлен результат выполнения программы.

```
Значение переменной x внутри метода f(int): 10
Значение переменной x внутри метода f(double): 10,1
Значение переменной x внутри метода f(int): 99
Значение переменной x внутри метода f(int): 10
Значение переменной x внутри метода f(double): 11,5
```

В этом примере определены только две версии метода `f()`. В одной из них параметр имеет тип `int`, а в другой — `double`. Однако в C# существует возможность передать методу `f()` значение типа `byte`, `short` или `float`. При передаче методу значения типа `byte` или `short` C# автоматически преобразует его в тип `int`. При передаче значения типа `float` его тип преобразуется в `double` и вызывается метод `f(double)`.

Важно понимать, что автоматическое преобразование типов применяется только при несовпадении типов параметра и аргумента. А вот еще одна версия предыдущей программы, в которую добавлен метод `f()` с параметром типа `byte`.

```
// Еще одна версия предыдущей программы, в которую добавлен метод f(byte).

using System;

class Overload2 {
  public void f(byte x) {
    Console.WriteLine("Значение переменной x внутри метода f(byte): " + x);
  }

  public void f(int x) {
    Console.WriteLine("Значение переменной x внутри метода f(int): " + x);
  }

  public void f (double x) {
    Console.WriteLine("Значение переменной x внутри метода f(double): " + x);
  }
}

class TypeConv {
  public static vold Main() {
    Overload2 ob = new Overload2();

    int i = 10;
    double d = 10.1;
    byte b = 99;
    short s = 10;
    float f = 11.5F;

    ob.f(i); // Вызывается метод ob.f(int)
    ob.f(d); // Вызывается метод ob.f(double)
```

```
    ob.f(b);  // Вызывается метод ob.f(byte). Теперь автоматическое
              // преобразование типа не выполняется.

    ob.f(s);  //  Вызывается  метод ob.f(int) и выполняется автоматическое
              // преобразование типа аргумента.
    ob.f(f);  //  Вызывается  метод ob.f(double) и выполняется автоматическое
              // преобразование типа аргумента.
  }
}
```

Результат выполнения этой программы отличаются от предыдущего только тем, что в третьей строке упоминается метод `f(byte)`, а не метод `f(int)`.

```
Значение переменной x внутри метода f(int): 10
Значение переменной x внутри метода f(double): 10,1
Значение переменной x внутри метода f(byte): 99
Значение переменной x внутри метода f(int): 10
Значение переменной x внутри метода f(double): 11,5
```

Так как существует версия метода, `f(byte)`, при вызове метода `f()` с аргументом типа `byte` вызывается метод `f(byte)` без преобразования типа значения аргумента в `int`.

При перегрузке методов также могут использоваться модификаторы `ref` и `out`. В следующем фрагменте кода определены два разных метода:

```
public void f(int x) {
    Console.WriteLine("Значение переменной x внутри метода f(int): " + x);
}

public void f(ref int x) {
    Console.WriteLine("Значение переменной x внутри метода f (ref int): " + x);
}
```

Таким образом, оператор

```
ob.f(i);
```

вызывает метод `f(int x)`, но оператор

```
ob.f(ref i);
```

вызывает метод `f(ref int x)`.

Перегрузка методов поддерживает полиморфизм, поскольку является одним из способов реализации в C# принципа «один интерфейс, множество методов». В тех языках, которые не поддерживают перегрузку методов, каждому методу необходимо присвоить уникальное имя. Но довольно часто возникает ситуация, когда требуется применить один метод для различных типов данных. Рассмотрим функцию возврата абсолютного значения. В языках, не поддерживающих перегрузку, имеется три и более версий такой функции с немного отличающимися именами. Так, в C функция `abs()` возвращает абсолютное значение целого числа, функция `labs()` — абсолютное значение длинного целого числа, а функция `fabs()` — абсолютное значение числа с плавающей точкой. Поскольку в C перегрузка не поддерживается, каждая функция должна иметь собственное имя, хотя эти функции аналогичны. Такой подход очень усложняет работу: несмотря на то, что основные действия, выполняемые каждой функцией, одинаковы, программисту все равно необходимо помнить три имени. В C# подобная ситуация исключена, так как все методы, предназначенные для вычисления абсолютного значения, могут иметь одно имя. Стандартная библиотека классов в C# содержит метод `Abs()`, возвращающий абсолютное значение. Он

перегружается классом System.Math для обработки всех числовых типов. А компилятор, исходя из типа аргумента, определяет, какую версию метода Abs () следует вызвать.

Преимущество перегрузки заключается в том, что она позволяет, используя общее имя, получать доступ к методам, выполняющим одинаковые действия с различными пшами данных. Таким образом, имя Abs представляет *общее выполняемое действие.* Выбор необходимой версии метода для указанного типа данных производится компилятором, программист же обязан помнить лишь общее действие, выполняемое данным методом. Благодаря реализации полиморфизма вместо нескольких имен используется только одно. Этот пример с методом AЬc () достаточно простой, но использование перегрузки позволяет решить гораздо более сложные задачи.

Ответы профессионала

Вопрос. Мне встречался термин *подпись*, который используют программисты ни С. Что он означает?

Ответ. В C# подпись — это имя метола со списком его параметров. Следовательно, при перегрузке двух методов одного и того же класса они не могут иметь одинаковую подпись. Обратите внимание, что подпись не включает тип возвращаемого значения, поскольку он tic используется компилятором C# для различия перегруженных методов.

При перегрузке метода каждая из версий может выполнятьлюбыс необходимые действия. Нет правила, которое бы определяло, что перегруженные методы должны выполнять одинаковые или похожие действия. Но стиль написания программ в C# предполагает, что перегрузка методов должна быть связана с предыдущей версией метода. Следовательно, хотя у вас есть возможность использовать одно имя дли перегрузки методов, выполняющих в результате абсолютно разные действия, делать этого не следует. Так, можно использовать имя sqr для создания метола, возвращающею *квадрат* целочисленного значения, и метода, возвращающего *квадратный корень* значения с плавающей точкой. Но создав два принципиально различных ме тода, программист только усложнит понимание программы и увеличит вероятность появления ошибок. Поэтому перегружать рекомендуется только тесно связанные операции.

Минутный практикум

1. Какое условие должно выполняться, чтобы метод мог быть перегружен?
2. Необходимо ли перегруженному методу иметь тип возвращаемого значения, отличающийся от предыдущем версии?
3. Как выполняется автоматическое преобразование типов данных в C# при перегрузке?

1. Для перегрузки метод должен отличаться от своей предыдущей версии по типу и/или количеству параметров.

2. Нет. Тип возвращаемого значения перегруженных методов может отличаться, но это не является необходимым условием для перегрузки метода.

3. Когда отсутствует прямое совпадение типов между набором аргументов и набором параметров, используется метод с наиболее близко совпадающим набором аргументов, если типы этих аргументов могут быть автоматически преобразованы к типам, имеющим параметры.

Перегрузка конструкторов

Также как методы, конструкторы могут быть перегружены, что позволяет конструировать объекты различными способами. В качестве примера рассмотрим следующую программу:

```
// В программе демонстрируется использование перегруженного конструктора.

using System;

 class MyClass {
   public int x;

  public MyClass()  {
    Console.WriteLine("Этот оператор определен" +
                      " в конструкторе MyClass().");
    x = 0;
  }

  public MyClass(int i) {
    Console.WriteLine("Этот оператор определен" +
                      " в конструкторе MyClass(int).");
    x = i;
  }

  public MyClass(double d) (
    Console.WriteLine("Этот оператор определен" +
                      " в конструкторе MyClass(double) .") ;
    x = (int) d;
  }

  public MyClass(int i, int j) {
    Console.WriteLine("Этот оператор определен" +
                      " в конструкторе MyClass(int, int).");
    x i * j;
  }
}

class OverloadConsDemo {
  public static void Main() {
    MyClass tl = new MyClass();
    MyClass t2 = new MyClass(88);
    MyClass t3 = new MyClass(17.23);
    MyClass t4 = new MyClass(2, 4);

    Console.WriteLine("Значение переменной t1.x: " + t1.x);
    Console.WriteLine("Значение переменной t2.x: " + t2.x);
    Console.WriteLine("Значение переменной t3.x: " + t3.x);
    Console.WriteLine("Значение переменной t4.x: " + t4.x);
  }
}
```

Конструирование объектов различными способами.

Результат выполнения программы таков:

```
Этот оператор определен в конструкторе MyClass().
Этот оператор определен в конструкторе MyClass(int).
Этот оператор определен в конструкторе MyClass(double).
```

```
  Этот оператор определен в конструкторе MyClass(int, int).
Значение переменной t1.x: 0
Значение переменной t2.x: 88
Значение переменной t3.x: 17
Значение переменной t4.x: 8
```

Конструктор `MyClass()` может быть перегружен четырьмя способами, при использовании каждого из них объект конструируется по-разному. Выбор компилятором необходимого конструктора осуществляется на основе параметра, указанного при выполнении оператора new. Благодаря перегрузке конструктора появляется возможность гибкого выбора способа конструирования объекта и создания именно такого объекта, который необходим в конкретной ситуации.

Использование перегрузки позволяет конструктору одного объекта инициализировать другой объект. Например, рассмотрим программу, в которой используется класс `Summation` для вычисления суммы всех чисел от ноля до значения, указанного в качестве аргумента конструктора этого класса.

```
// В этой программе один объект используется для инициализации другого.

using System;

class Summation {
  public int sum;

  // Для создания объекта конструктор использует параметр типа int.
  public Summation(int num) {
    sum = 0;
    for(int i=1; i <= num; i++)
      sum += i;
  }

  // Для инициализации объекта данный конструктор использует другой объект,
  // переданный ему в качестве параметра.
  public Summation(Summation ob) {
    sum = ob.sum;
  }
}

class SumDemo {
  public static void Main() {
    Summation s1 = new Summation(5);
    Summation s2 = new Summation(s1);

    Console.WriteLine("Сумма чисел от 0 до 5 (значение переменной s1.sum) = "
            + s1.sum);
    Console.WriteLine("Значение переменной s2.sum: " + s2.sum);
  }
}
```

Один объект используется для инициализации другого объекта.

Результат выполнения программы показан ниже.

```
Сумма чисел от 0 до 5 (значение переменной s1.sum) = 15
Значение переменной s2.sum: 15
```

Как видно из приведенного выше примера, использование одного элемента для создания другого позволяет гораздо эффективнее выполнять инициализацию создаваемого объекта. В данном случае при конструировании объекта `s2` отсутствует необходимость повторно вычислять сумму всех чисел от 0 до 5.

Вызов перегруженного конструктора с использованием ключевого слова this

При вызове перегруженного конструктора в некоторых случаях бывает полезно использовать вызов одного конструктора другим. В C# это реализуется при помощи еще одной формы ключевого слова this. Общий синтаксис такого конструктора представлен ниже:

```
constructor-name (parameter-list) : this (argument-list) {
   //  ... тело конструктора, который также может быть пустым
}
```

При вызове этого конструктора (который можно назвать вызывающим) первым выполняется перегруженный конструктор, в котором список параметров совпадает со *списком аргументов*. Далее выполняются операторы, определенные внутри вызывающего конструктора, если таковые имеются. Ниже представлена программа, где используется ключевое слово this.

```
// В программе демонстрируется использование ключевого слова this для вызова
// одного конструктора из другого.

using System;

class XYCoord {
  public int x, y;

  public XYCoord() : this(0, 0) {
    Console.WriteLine("Этот оператор определен в конструкторе XYCoord()");


  public XYCoord(XYCoord obj) : this(obj.x, obj.y) {
    Console.WriteLine("Этот оператор определен в конструкторе " +
                      "XYCoord(XYCoord obj)");
   }

  public XYCoord(int i, int j) {
    Console.WriteLine("Этот оператор определен в конструкторе " +
                      "XYCoord(XYCooro(int, int)");
    x = i;
    y = j;
   }

}
class OverloadConsDemo {
  public static void Main() {
    XYCoord t1 = new XYCoord();
    XYCoord t2 = new XYCoord(8, 9);
    XYCooro t3 = new XYCoord(t2);

    Console.WriteLine("Значение переменных t1.x и tl.y: " + t1.x + ", "
             + t1.y);
    Console.WriteLine("Значение переменных t2.x и t2.y: " + t2.x + ", "
             + t2.y)
    Console.WriteLine("Значение переменных t3.x и t3.y: " + t3.x + ", ”
             + t3.y);
   }
}
```

В результате выполнении этой программы на экран будут выведены следующие строки:

```
     Этот оператор определен в конструкторе XYCoord(XYCoora(int, int)
     Этот операторопределен в конструкторе XYCoord()
Этот оператор определен в конструкторе XYCoora(XYCoord(int, int)
Этот оператор определен в конструкторе XYCoord(XYCoora(int, int)
Этот оператор определен в конструкторе XYCoord(XYCoord obj)
Значение переменных t1.x и t1.y: 0, 0
Значение переменных t2.x и t2.y: 8, 9
Значение переменных t3.x и t3.y: 8, 9
```

рассмотрим, как работает эта программа. В классе `XYCoord` единственным конструктором, который фактически инициализирует поля `x` и `y`, является конструктор `XYCoord(int, int)`. Остальные два конструктора, используя ключевое слово `this`, просто вызывают конструктор `XYCoord(int, int)`. Так, при создании объекта `t1` вызывается его конструктор `XYCoord()`, который в свою очередь вызывает конструктор `this(0, 0)`. То есть ключевое слово `this` указывается вместо полного имени конструктора `XYCoord(0, 0)`. Создание объекта `t2` происходит таким же образом.

Одна из причин использования ключевого слова `this` для вызова перегруженных конструкторов — возможность избежать дублирования кода. В предыдущем примере без использования ссылки `this` пришлось бы повторять последовательность инициализирующих операторов (`x = i; y = j;`) но всех трех конструкторах. Еще одним преимуществом применения ключевого слова `this` является создание конструкторов с «аргументами по умолчанию», которые применяются, если эти аргументы не заданы явно. В частности, для предыдущей программы можно создать еще один конструктор `XYCoord()`:

```
public XYCoord(int x) : this(x, x) {}
```

который автоматически присваивает координате `y` то же значение, что и координате `x`. Учтите, что «аргументы по умолчанию» необходимо использовать осторожно, поскольку их неправильное определение может привести к ошибкам при выполнении программы.

Проект 6-2. Перегрузка конструктора `Queue`

`QDemo2.cs`

В этом проекте мы модернизируем класс `Queue` путем добавления к нему двух конструкторов. С помощью первого конструктора из одной очереди будет создана Другая очередь. Второй конструктор мы используем для создания очереди и присваивания начальных значений ее элементам (инициализации). Добавление этих двух конструкторов позволит использовать данный класс более эффективно.

Пошаговая инструкция

1. Создайте новый файл с именем `QDemo2.cs` и скопируйте в него код класса `Queue` из проекта 6-1.
2. Добавьте в программу представленный ниже конструктор, который создает очередь из другой очереди.

```
// Конструктор создает объект типа Queue на основе другого объекта типа Queue.
public Queue(Queue ob) {
```

```
    putloc = ob.putloc;
    getloc = ob.getloc;
    q = new char[ob.q.Length];

    // Копирование элементов,
    for(int i=getioc+1; i <= putloc; i++)
      q[i] = ob.q[i];
  }
```

Давайте внимательно рассмотрим этот конструктор. Он присваивает переменным `putloc` и `getloc` начальные значения, содержащиеся в параметре (объекте) `ob`. Затем он создает новый массив для хранения элементов очереди и копирует их из массива объекта `ob` в этот массив. После конструирования новая очередь будет точной копией начальной (используемой для инициализации) очереди, но обе они будут совершенно разными объектами.

3. Добавьте конструктор, который инициализирует очередь из массива символов, передаваемого в качестве параметра.

```
// Конструктор создает объект типа Queue и инициализирует его.
public Queue (char[] a) {
  putloc = 0;
  getloc = 0;
  q = new char[a.Length+1];

  for(int i = 0; i < a.Length; i++) put(a[i]);
}
```

Этот конструктор создает очередь, длина которой позволяет хранить все символы из массива `a`, а затем помещает эти символы в массив очереди. Для корректной работы алгоритма очереди ее длина (длина ее массива) должна быть на единицу больше длины массива, передаваемого в качестве параметра.

4. Ниже представлен полный код обновленного класса `Queue` вместе с кодом класса `QDemo2`.

```
// Версия класса Queue с двумя дополнительными конструкторами.

 using System;

class Queue {
  // Теперь эти члены класса по умолчанию становятся закрытыми.
  char[] q;              // Этот символьный массив является основой очереди.
  int putloc, getloc;    // Индексы, указывающие, какому элементу массива
                         // будет присваиваться значение и из какого
                         // элемента значение будет считываться.

  // Создает пустую очередь по заданному размеру.
  public Queue(int size) {
    q = new char[size+1];     // Выделение памяти для очереди,
    putloc = getloc = 0;
  }

// Конструктор создает о&ьект типа Queue на основе другого объекта типа Queue.
  public Queue(Queue ob) {
    putloc = ob.putloc;
    getloc = ob.getloc;
    q = new char[ob.q.Length];
```

```
    // Копирование элементов,
    for(int i=getloc+1; i <= putloc; i++)
      q [i] = ob.q[i] ;
  }

  // Конструктор создает объект типа Queue и инициализирует его.
  public Queue(char[] a) {
    putloc = 0;
    getloc = 0;
    q = new char[a.Length+1];

    for(int i = 0; i < a.Length; i++) put(a[i]);
  }
  // Метод помещает символ в очередь.
  public void put(char ch) {
    if(putloc==q.Length-1) {
      Console.WriteLine("- Очередь заполнена.");
      return;
    }

    putloc++;
    q[putloc] = ch;
  }

  // Считывает значения элементов массива (очереди).
  public char get() {
    if(getloc == putloc) {
      Console.WriteLine("- Очередь пустая.");
      return (char) 0;
    }

    getloc++;
    return q[getloc];
  }
}
// В этом классе демонстрируется использование класса Queue,
class QDemo2 {
  public static void Main() {
    // Создается пустая очередь из десяти элементов.
    Queue q1 = new Queue(10);

    char[] name = {'T', 'o', 'm'};
    // Создается очередь на основе массива.
    Queue q2 = new Queue(name);

    char ch;
    int i;

    // В очередь помещаются некоторые символы.
    for(i=0; i < 10; i++)
      q1.put((char) ('A' + i));

    // Создается очередь на основе другой очереди.
    Queue q3 = new Queue(q1);

    // Отображение значений элементов очереди.
```

```
        Console.Write("Содержимое очереди q1: ");
        for(i=0; i < 10;  i++){
          ch = q1.get();
          Console.Write(ch);
        }

        Console.WriteLine("\n");

        Console.Write("Содержимое очереди q2: ");
        for(i=0; i < 3; i++)  {
          ch = q2.gee();
          Console.Write(ch) ;
        }

        Console.WriteLine("\n");

        Console.Write("Содержимое очереди q3: ");
        for (i=0; i < 10; i++) {
          ch = q3.get();
          Console.Write(ch);
        }
      }
    }
```

5. Ниже представлен результат выполнения программы.

```
Содержимое очереди q1: ABCDEFGHIJ

Содержимое очереди q2:  Tom

Содержимое очереди q3: ABCDEFGHIJ
```

Метод `Main()`

До настоящего момента мы использовали только одну версию метода `Main()`. Но в C# существует несколько перегруженных версий этого метода. Некоторые из них могут использоваться для возвращения значений, а другие могут принимать аргументы. Каждая из этих версий будет рассматриваться в данном разделе.

Возвращение значений методом `Main()`

При завершении работы программы метод `Main()` может возвратить значение вызывающему процессу (часто операционной системе). Для этого используется следующий синтаксис метода `Main()`:

```
public static int Main()
```

Обратите внимание, что вместо типа возвращаемого значения `void` в этой версии метода `Main()` указан тип `int`. Следовательно, возвращаемое этим методом значение будет иметь тип `int`.

Обычно возвращаемое значение метода `Main()` указывает способ завершения программы. По соглашению, если возвращается значение `0`, то программа завершена корректно. Все другие значения указывают на ошибку, возникшую при выполнении программы.

Передача аргументов методу `Main()`

Многие программы принимают так называемые *аргументы командной строки*. Аргумент командной строки — это информация, которая следует непосредственно за именем программы в командной строке. В Сопрограммах такие аргументы передаются методу `Main()`. Чтобы этот метод мог принимать аргументы, необходимо использовать следующий синтаксис:

```
public static void Main(string[] args)
```

или

```
public static int Main(string[] args)
```

В первом варианте метод `Main()` имеет тип возвращаемого значения `void`; второй вариант можно использовать для возвращения целочисленного значения, как уже описывалось в предыдущем разделе. В обоих случаях аргументы командной строки хранятся в виде строк в массиве типа `string`, который передастся методу `Main()`. Например, в следующей программе на экран выводятся все аргументы, с которыми эта программа вызывается.

```
// Программа отображает все аргументы, заданные в командной строке.

using System;
  class CLDemo {
    public static void Main(string[] args) {
                        Console.WriteLine("Программе были переданы " + args.Length +

          " аргумента командной строки.");
    Console.WriteLine("Список аргументов: ");
      for(int i=0; i<args.Length; i++)
  Console.WriteLine(args[i]);
  }
}
```

Если в командной строке ввести имя программы `CLDemo` и задать три аргумента

```
CLDemo один, два, три.
```

на экран будут выведены следующие строки:

```
Программе были переданы 3 аргумента командной строки.
Список аргументов:
Один,
два,

три.
```

Чтобы понять механизм использования аргументов командной строки, рассмотрим следующую программу. Она принимает один аргумент командной строки — имя пользователя. Затем производится поиск этого имени в двухмерном строковом массиве. Если такое имя найдено, на экран выводится телефонный номер этого пользователя.

```
// Простой автоматизированный телефонный справочник.

 using System;
  class Phone {
    public static int Main(string[] args) {
      string[,] numbers = {
      { "Владимир", "555-3322" },
      { "Людмила", "555-8976" },
```

```
        { "Сергей", "555-1037" },
        { "Александра", "555-1400" },
        { " ", "" }  // Определена нолевая строка для случая, когда не будет
                     // введено никакого имени.
      };
      int i;

      if(args.Length != 1) {
        Console.WriteLine("Шаблон ввода: Phone <имя>");
        return 1;  // Значение, указывающее на неправильный ввод аргумента и
                   // преждевременное завершение программы.
      }
      else {
        for(i=0; numbers[i, 0] != x++) {
          if(numbers[i, 0] == args[0]) {
            Console.WriteLine(numbers[i, 0] + ": " +
                              numbers[i, 1]);
            break;
          }
        }
        if(numbers[i, 0] == "")
          Console.WriteLine("Имя не найдено.");
      }
      return 0;
    }
  }
```

Ниже приведен пример работы этой программы.

```
D:\Work\C#\programs\ch_C6>Phone Людмила
Людмила: 555-8976
```

Рассмотрим данную программу подробнее. Во-первых, заметьте, что перед продолжением выполнения программы проверяется наличие аргумента командной строки. Это очень важный момент. Если корректная работа программы зависит от количества введенных аргументов, всегда необходимо вначале проверить количество аргументов командной строки, передаваемое такой программе. Невыполнение этого условия часто приводит к аварийному завершению работы программы.

Во-вторых, обратите внимание на то, как программа возвращает код завершения работы. Если программе не было передано требуемое количество аргументов, то возвращается значение 1, что указывает на аварийное завершение программы. Если программе было передано нужное число аргументов, то при завершении программы возвращается значение 0.

Минутный практикум

1. Может ли конструктор принять в качестве параметра объект своего класса?
2. Для чего применяются перегруженные конструкторы?
3. Покажите синтаксис метода `Main()`, который принимает аргументы командной строки, но не возвращает значение.

1.Да

2. Перегруженные конструкторы применяются для обеспечения удобства и гибкости при использовании классов.

3. `public static void Main(string[] args)`

Рекурсия

В C# метод может вызывать сам себя. Этот процесс называется рекурсией, а метод, который сам себя вызывает, называется рекурсивным. Ключевым компонентом рекурсивного метода является оператор для вызова этого же метода.

Классическим примером рекурсии является вычисление факториала числа. Факториалом числа N называется произведение всех целых чисел от 1 до N (скажем, факториал числа 3 равен произведению 1x2x3 = 6). В следующей программе представлен рекурсивный метод, предназначенный для вычисления факториала числа. Для сравнения в программу включен равнозначный метод, выполняющий те же функции без использования рекурсии.

```
// В программе демонстрируется использование рекурсии для вычисления
// факториала числа.

using System;

class Factorial {
  // Рекурсивный метод,
  public int factR(int n) {
    int result;

    if (n==1} return 1;
    result = factR(n-l) * n;
    return result;
  }

  // Равнозначный метод, в котором вместо рекурсии используется цикл for.
  public int factl(int n) {
    int t. result;

    result = 1;
    for(t=l; t <= n; t++) result *= t;
    return result;
  }
}
class Recursion {
 public static void Main() {
    Factorial f = new Factorial();

    Console.WriteLine("Факториал числа, вычисленный с помощью рекурсивного" +
                      " метода. ");
    Console.WriteLine("Факториал числа 3 = " + f.factR(3));
    Console.WriteLine("Факториал числа 4 = " + f.factR(4));
    Console.WriteLine("Факториал числа 5 = " + f.factR(5));
    Console.WriteLine();

    Console.WriteLine("Факториал числа, вычисленный с помощью обычного " +
                      "метода.");
    Console.WriteLine("Факториал числа 3 = " + f.factl(3));
    Console.WriteLine("Факториал числа 4 = " + f.factl(4));
    Console.WriteLine("Факториал числа 5 = " + f.factl(5));
  }
}
```

Рекурсивный вызов метода `factR()`.

Ниже показан результат выполнения этой программы.

```
Факториал числа, вычисленный с помощью рекурсивного метода.
Факториал числа 3 = 6
Факториал числа 4 = 24
Факториал числа 5 = 120

Факториал числа, вычисленный с помощью обычного метода.
Факториал числа 3 = 6
Факториал числа 4 = 24
Факториал числа 5 = 120
```

Алгоритм работы обычного метода `fact()` достаточно прост. В нем используется цикл, где последовательно умножаются все целые числа от 1 до числа, факториал которого нужно вычислить.

Работа рекурсивного метода `factR()` немного сложнее. Если вызвать метод `factR()` с аргументом `1`, метод возвращает единицу, которая при возврате управления предыдущему методу `factR()` будет использоваться в выражении `factR(n-1)*n`. При выполнении этого выражения метод `factR()` вызывается с аргументом `n-1`. Этот процесс повторяется до тех пор, пока значение переменной `n` не будет равно `1`. Так, при вычислении факториала числа 2 первый вызов метода `factR()` с аргументом `2` приведет ко второму вызову этого же метода, но уже с аргументом `1`. При получении этого аргумента метод возвратит число 1, которое затем будет умножено на 2. То есть ответ — значение 2. Можно поместить оператор `WriteLine();` в метол `factR()`, который будет выводить на экран аргумент каждого вызова и промежуточные результаты.

Когда метод вызывает сам себя, в стек помещаются новые локальные переменные и параметры, а код метода выполняется с этими новыми переменными с самого начала. Рекурсивные вызовы не создают новую копию метода, новыми являются только аргументы. После возвращения рекурсивным методом значения старые локальные переменные и параметры удаляются из стека, а управление программой передается оператору, следующему за тем, из которого был в вызван метод.

Рекурсивные версии многих программ могут выполняться медленнее своих итеративных (использующих обычные методы и циклы) аналогов, так как при дополнительных вызовах метода появляются непроизводительные издержки. В результате слишком большого количества рекурсивных вызовов создается много новых локальных переменных и параметров, а это может перегрузить стек, что приведет к возникновению исключительной ситуации. Обычно это происходит, если в программе неправильно определено условие: метод вместо вызова самого себя должен вернуть значение.

Основное преимущество рекурсии заключается в том, что некоторые алгоритмы рекурсивно могут быть реализованы проще и эффективнее, чем их итеративный вариант (например, алгоритм быстрой сортировки достаточно сложен при его итеративной реализации). Рекурсивные решения требуются и при решении некоторых задач, связанных с разработкой искусственного интеллекта.

Как правило, для возврата из метода значения без выполнения рекурсивного вызова используется условный оператор, такой как `if`. Учтите, что при неправильном определении условного выражения после вызова метод будет бесконечно вызывать сам себя. У начинающих программистов такая ошибка при использовании рекурсии встречается достаточно часто. Мы рекомендуем для контроля над выполнением программы как можно чаще использовать операторы `WriteLine();` и прерывать программу в случае обнаружения ошибки.

Ключевое слово static

Иногда требуется определить член класса, который будет использоваться независимо от любых объектов этого класса. Обычно доступ к члену класса осуществляется с использованием объекта этого класса, но существует возможность создать член класса, который может применяться самостоятельно без ссылки на созданный объект какого-либо класса. При объявлении такого члена класса используется ключевое слово static. Если член класса определяется с модификатором static, он становится доступным до создания объектов этого класса и без ссылки на какой-либо объект. Ключевое слово static может использоваться для объявления как методов, так и переменных. Наиболее известным членом класса с модификатором static является метод Main(), который объявляется как static, поскольку должен вызываться операционной системой при запуске программы.

Для использования члена класса с модификатором static за пределами класса следует указать имя его класса, за которым следует оператор точка (.). В этом случае создавать объект нет необходимости. Фактически доступ к члену класса с модификатором static нельзя осуществить, указав идентификатор объекта, доступ к нему осуществляется только с использованием имени класса. Например, если переменной static count, являющейся частью класса Timer, необходимо присвоить значение 10, нужно использовать следующий оператор:

```
Timer.count = 10;
```

Этот синтаксис аналогичен синтаксису, используемому для доступа к обычной переменной экземпляра, за исключением того, что вместо идентификатора объекта применяется имя класса. Метод с модификатором static также вызывается с использованием имени его класса и оператора точка.

Переменная, объявленная с модификатором static, фактически является глобальной переменной. При объявлении объектов ее класса не создастся копия такой переменной, вместо этого все объекты класса совместно используют одну переменную с модификатором static, которая инициализируется при загрузке ее класса. Если начальное значение не указано явно, по умолчанию такой переменной присваивается значение 0 (если это переменная числового типа), значение null (если это переменная ссылочного типа) и значение false (если это переменная типа bool). Таким образом, переменная с модификатором static всегда имеет значение.

Различие между методом с модификатором static и обычным методом состоит в том, что метод с этим модификатором может быть вызван с использованием имени его класса без создания какого-либо объекта этого метода. Ранее нам уже встречался пример такого вызова — метод Sqrt() с модификатором static класса System.Math.

Ниже представлена программа, в которой создаются переменная и метод с модификаторами static.

```
   // В программе демонстрируемся использование модификатора static.

      using System;

class StaticDemo {
  // Объявление статической переменной.
  public static int val = 100;

  // Объявление статического метода.
```

```
  public static int valDiv2() {
    return val/2;
  }


class SDemo {
  public static void Main() {

    Console.WriteLme("Первоначальное значение переменной StaticDemo.val = "
                        + StaticDemo.val);

    StaticDemo.val = 8;
    Console.WriteLine("Значение переменной StaticDemo.val = " +
                StaticDemo.val);
    Console.WriteLine{"Значение этой же переменной после выполнения метода " +
                        "StaticDemo.valDiv2() = " + StaticDemo.valDiv2());
  }
}
```

В результате работы этой программы будут выведены следующие строки и значения:

```
Первоначальное значение переменной StaticDemo.val = 100
Значение переменной StaticDemo.val = 8
Значение этой же переменной после выполнения метода StaticDemo.valDiv2() = 4
```

Как видно из результата работы программы, переменная с модификатором `static` инициализируется в начале работы программы до создания какого-либо объекта ее класса.

Методы с модификатором `static` обладают некоторыми ограничениями:

- метод с модификатором `static` не имеет ссылки `this`;
- метод, объявленный как `static`, может непосредственно вызвать только другой статический метод (метод с модификатором `static`). Он не может вызвать метод экземпляра своего класса, потому что методы экземпляра класса могут работать с данными экземпляров этого класса, а статические методы не могут;
- метод с модификатором `static` имеет прямой доступ только к статическим данным, он не может использовать переменную экземпляра, поскольку не работает с экземплярами своего класса.

Например, в приведенном ниже фрагменте метод `valDivDenom()` с модификатором `static` является недействительным:

```
class StaticError {
  int denom = 3; // обычная переменная экземпляра.
  static int val = 1024; // Статическая переменная.

  /* Ошибка! Статический метод не может непосредственно обращаться к
     обычной переменной.
   */
  static int valDivDenom() {
    return val/denom; // Этот оператор не будет скомпилирован.
  }
}
```

Обычная переменная экземпляра, доступ к которой не может быть осуществлен из метода с модификатором `static`, указывается с помощью параметра `denom`. Однако использование переменной `val` разрешено, поскольку это переменная статическая.

В следующей программе та же проблема возникает при попытке вызова обычного метода `nonStaticMeth()`, определенного в классе `AnotherStaticError`, из метода `staticMeth()`, объявленного как `static`, и гоже определенного в этом классе.

```
using System;

class AnotherStaticError {
  // Обычный метод.
  void nonStaticMeth()  {
     Console.WriteLine("Оператор метода nonStaticMeth().");


  /* Ошибка! Статический метод не может непосредственно обращаться к
     обычному.
   */
  static void staticMeth() {
    nonStaticMeth();   // Этот оператор не будет скомпилирован.
  }
}
```

В этом случае попытка вызова обычного метода (то есть метода экземпляра) из метода с модификатором `static` приводит к ошибке компилирования программы.

Обратите внимание, что метод с модификатором `static` *может* вызывать методы экземпляра и получать доступ к переменным экземпляра своего класса, но осуществлять это необходимо, используя объект данного класса (то есть для доступа к методу или переменной экземпляра нужно указывать название объекта). Приведенный ниже фрагмент кода является действительным.

```
class MyClass {
  // Обычный метод.
  void nonStaticMeth()  {
     Console.WriteLine("Оператор метода nonStaticMeth().");
   }

  /* Используя переменную ссылочного типа, можно вызвать обычный метод кз
     статического метода.
   */
  public static void staticMeth(MyClass ob) {
    ob.nonStaticMeth(); // Это действительный вызов метода.

   }
}
```

Минутный практикум

1. Дайте определение рекурсивного метода.
2. Объясните различие между переменными с модификаторами `static` и переменными экземпляра.
3. Может ли методе модификатором `static` вызывать обычный метод одного с ним класса без использования ссылки на объект.

1. Рекурсивным называется метод, который вызывает сам себя.

2. Каждый объект класса имеет свои собственные копии переменных экземпляра, определенные классом. Все объекты класса совместно используют одну копию переменной с модификатором `static`.

3. Нет.

Проект 6-3. Алгоритм Quicksort

QSDemo.cs

В главе 5 был представлен простой алгоритм сортировки, называемый пузырьковым, а также было сказано, что существуют более эффективные методы сортировки. В этом проекте мы разработаем версию одного из лучших способов сортировки Quicksort (алгоритм быстрой сортировки). Этот алгоритм не описывался в главе 5, поскольку его работа основана на рекурсии. Версия, которую мы будем разрабатывать, предназначена для сортировки символьного массива, но принцип работы этого алгоритма может быть применен для сортировки объектов любого типа.

Принцип работы алгоритма быстрой сортировки состоит в следующем — большой объем данных делится на более мелкие части, которые успешно и быстро обрабатываются. Поскольку алгоритм Quicksort применяется для сортировки элементов массива, то логично при делении этого объема данных на части использовать значение длины массива. Из значения крайнего правого индекса вычитается значение крайнего левого индекса и делится на 2. Полученный результат является индексом элемента (в дальнейшем мы будем называть его средней точкой), значение которого становится ориентиром на первом этапе сортировки. Все элементы, имеющие значения большие или равные значению средней точки, перемещаются в правую часть массива, а элементы, имеющие значения меньшие, чем значение средней точки, перемещаются в левую часть массива. То есть в результате выполнения первого этапа сортировки массив как бы делится на две части. Затем этот же процесс повторяется с каждой частью до тех пор, пока весь массив не будет отсортирован. Например, если в качестве средней точки используется элемент со значением `d`, то после первого этапа сортировки элементы символьного массива `f e d a c b` будут перераспределены следующим образом:

```
Первоначально                    f e d a cb
После первого этапа сортировки   b c a d ef
```

Затем этот процесс повторяется для каждой части массива, то есть для `b c a` и `d e f`.

Выбрать среднюю точку можно двумя способами — произвольной путем нахождения среднего значения небольшой группы значений, взятых из массива. Для оптимальной сортировки необходимо выбрать значение, находящееся в середине диапазона значений элементов массива, но для многих наборов данных это осуществить не очень просто, В худшем случае выбранное значение будет одним из крайних значений (либо минимальным, либо максимальным). Но даже в этом случае алгоритм Quicksort будет работать корректно. В разрабатываемой нами версии программы в качестве средней точки выбирается средний элемент массива.

Пошаговая инструкция

1. Создайте файл и назовите его `QSDemo.cs`.
2. Создайте класс `Quicksort`, как показано ниже.

```
// Простая версия алгоритма Quicksort,

using System;

class Quicksort {
```

```
  // Метод предназначен для вызова другого метода qs, в котором
  // непосредственно реализован алгоритм Quicksort.
  public static void qsort(char[] items) {
    qs(items, 0, items.Length—1);
  }

   // рекурсивная версия алгоритма Quicksort, предназначенная для сортировки
  // символьного массива.
  static void qs(char[]; items, int left, int right)
  }
    int i, j;
    char x, y;

    i = left; j = right;
    x = items[(left+rignt)/2];

    do {
      while((items[i] < x) && (i < right)) i++;
      while((x < items[j]) && (j > left)) }--;

      if(i <= j) {
        y = items[i];
        items[i] = items[j];
        items[j] = y;
        i++; j--;
       }
    } while (i <= j);

    if(left < j) qs(items, left, j);
    if(i < right) qs(items, i, right);
  }
}
```

Для упрощения использования класса `Quicksort` в нем объявлен метод `qsort()`, предназначенный для определения параметров и вызова метода `qs()`, который непосредственно выполняет сортировку элементов. Это позволяет вызывать метод `qsort()`, указывая только имя массива, который необходимо отсортировать. Поскольку метод `qs()` используется только внутри класса, по умолчанию он определяется как `private`.

3. Чтобы использовать класс `Quicksort`, нужно вызвать метод `Quicksort.qsort()`. Поскольку метод `qsort()` определен как `static`, он может вызываться с использованием имени его класса, а не объекта, следовательно, нет необходимости создавать объект типа `Quicksort`. После возвращения методом значения массив будет отсортирован. Помните, что эта версия программы предназначена для сортировки только символьного массива, но принцип работы алгоритма можно использовать для сортировки массива любого типа.
4. Ниже представлена программа, демонстрирующая работу алгоритма `Quicksort`.

```
 // Простая версия алгоритма Quicksort,

using System;

class Quicksort {

  // Метод предназначен для вызова другого метода qs, в котором
```

```
  // непосредственно реализован алгоритм Quicksort,
  public static void qsort(char[] items) {
    qs(items, 0, items.Length-1);
  }

  // Рекурсивная версия алгоритма Quicksort, предназначенная для сортировка
  // символьного массива.
  static void qs(char[] items, int left, int right)
  {
    int i, j;
    char x, y;

    i = left; j = right;
    x = items[ (left + right)/2];

    do {
      while((items[i] < x) && (i < right)) i++;
      while((x < items[j]) && (j > left)) j--;

      if(i <= j) {
        y = items[i];
        items[i] = items[j];
        items[j] = y;
        i++; j--;
      }
    }   while(i <= j);

    if(left < j) qs(items, left, j);
    if(i < right) qs(items, i, right);
  }
}

class QSDemo {
  public static void Main() {
    char[] a = { 'd', 'x', 'a', 'r', 'p', 'j', 'i' };
    int i;

    Console.Write("Первоначальный массив: ");
    for(i=0; i < a.Length; i++)
      Console.Write(a[i]);

    Console.WriteLine();

    // Выполняется сортировка массива.
    Quicksort.qsort(a);

    Console.Write("Отсортированный массив: ");
    for(i=0; i < a.Length; i++)
      Console.Write(a[i]);
  }
}
```

Контрольные вопросы

1. Используя фрагмент кода

```
class X {
    int count;
```

определите, является ли корректным следующий код:

```
class Y {
    public static void Main() {
        X ob = new X();

        ob.count = 10;
```

2. Где должен располагаться модификатор при объявлении члена класса?
3. В стеке используется принцип LIFO (last in, first out — «последним зашел, первым вышел»). Стек часто сравнивается со стопкой тарелок — тарелка, первой поставленная на стол, будет использована последней. Создайте стек с именем `Stack` для хранения символов. Методы, осуществляющие доступ к стеку, назовите `push()` и `pop()`. Необходимо дать пользователю возможность указывать размер стека при его создании. Все остальные члены класса `Stack` должны иметь закрытый доступ (`private`). Подсказка: в качестве модели используйте класс `Queue`, в котором необходимо только изменить способ доступа к данным.
4. Задан следующий класс:

```
class Test {
    int a;
    Test(int i) { a = i; }
}
```

Напишите метод (`swap()`), меняющий местами содержимое двух объектов типа `Test`, на которые ссылаются две переменные ссылочного типа.

5. Является ли корректным приведенный ниже фрагмент кода?

```
class X {
    int meth(int a, int b) { ... }
    string meth(int a, int b) { ... }
```

6. Напишите рекурсивный метод, предназначенный для вывода строки в обратном порядке.
7. Как объявлять переменную, которую должны совместно использовать все объекты класса?
8. Какие функции выполняют модификаторы параметров `ref` и `out`? Чем они отличаются?
9. Напишите четыре варианта синтаксиса метода `Main()`.
10. Какие вызовы будут действительными для заданного фрагмента кода?

```
void meth(int i, int j, params int [] args) { // ...
```

а) `meth(10, 12, 19)`

б) `meth(10, 12, 19, 100)`

в) `meth(10, 12, 19, 100, 200)`

г) `meth(10, 12)`

Перегрузка оператора, индексаторы и свойства

- Основные сведения о перегрузке оператора
- Перегрузка бинарных операторов
- Перегрузка унарных операторов
- Перегрузка операторов сравнения
- Применение индексаторов
- Создание свойств

В главе рассматриваются три специальных типа членов класса: перегружаемые операторы, индексаторы и свойства. С их помощью можно создавать типы классов, аналогичные встроенным типам.

Перегрузка операторов

В C# можно определять действия, выполняемые оператором по отношению к какому-нибудь классу, который вы создаете. Это называется *перегрузкой или переопределением оператора.* Функции оператора можно полностью контролировать, но следует учитывать, что они различаются в зависимости от того, к какому классу применяется оператор. Например, в классе, определяющем связный список, оператор + используется для добавления объекта к списку, в классе, реализующем стек, этот оператор применяется для помещения объекта в стек, в других классах он может выполнять иные функции.

При выполнении перегрузки исходное предназначение оператора не теряется. Просто к его арсеналу добавляются новые операции, которые будут применяться по отношению к определенному классу. Поэтому, например, перегрузка оператора + для обработки связного списка не приведет к потере его исходного предназначения — выполнению операции сложения целых чисел.

Основным преимуществом использования перегрузки оператора является то, что при этом в программную среду можно легко интегрировать новый тип класса. Если для класса определены операторы, то действия с объектами этого класса можно выполнять, используя обычный синтаксис (то есть вместо применения отдельного метода, предназначенного для работы с переменными этого класса, достаточно ввести, например, такую строку кода: *объект + объект;)*. Кроме того, объекты можно использовать в выражениях, содержащих данные других типов.

Процесс перегрузки операторов очень похож на процесс перегрузки метода. Перегружая оператор, вы используете ключевое слово operator, определяющее *метод оператора*, который задает его действие.

Синтаксис метода оператора

В основном в C# используются два вида операторов — унарные и бинарные (операторы, для которых указываются два операнда). Для их перегрузки применяют два вида методов оператора, синтаксис которых представлен ниже.

```
// Синтаксис метода, используемого для перегрузки унарного оператора.
public static ret-type operator op(param-type operand)
{
// операции
}

// Синтаксис метода, используемого для перегрузки бинарного оператора.
public static ret-type operator op(param-cype1 operand1, param-type1
operand2)
{
// операции

}
```

Перегружаемый оператор (например, + или /) подставляется вместо слова op. Вместо словосочетания ret-type указывается тип значения, возвращаемого при выполнении данного метода. Это значение может принадлежать к любому выбранному пользователем типу, но обычно оно имеет тип класса, для которого выполняете перегрузка оператора. Поэтому перегружаемые операторы могут использоваться в выражениях. Методам, переопределяющим унарные операторы, операнды передаются с помощью параметра operand, а переопределяющим бинарные операторы, — помощью параметров operand1 и operand2.

Для перегрузки унарных операторов операнд должен принадлежать к типу класса для которого определяется оператор. Для перегрузки бинарных операторов тип как минимум одного из операндов должен совпадать с типом класса, для которого переопределяется оператор. Таким образом, в C# невозможно перегружать оператор для тех объектов, которые были созданы не вами. Например, нельзя переопределит оператор + для типа int или string. Кроме того, параметры оператора не должны включать модификаторы ref или out.

Перегрузка бинарных операторов

Чтобы продемонстрировать использование описанного выше процесса перегрузки оператора, рассмотрим программу, в которой перегружаются два бинарных оператора + и −. Вначале создается класс ThreeD, содержащий координаты объекта в трехмерном пространстве. Перегруженный оператор + выполняет сложение соответствующих координат двух объектов класса ThreeD. Перегруженный оператор − вычитает координаты одного объекта из соответствующих координат другого объекта.

```
// В программе демонстрируется использование перегруженных операторов.
using System;

// В классе содержатся три переменные x,y,z - координаты объекта.
class ThreeD {
  int x, y, z; // Координаты объекта.

  public ThreeD( ) {x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i;  y = j; z = k; }

  // Перегрузка бинарного оператора +.
  public static ThreeD operator +(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    /* Сложение значений двух соответствующих координат и возвращение
       результата. */
    result.x = op1.x + op2.x; // В этих трех строках кода выполняется
    result.y = op1.y + op2.y; // операция сложения целых чисел,
    result.z = op1.z + op2.z; // для которых, оператор + сохраняет
                              // свое исходное предназначение.

    return result;
  }

  // Перегрузка бинарного оператора
  public static ThreeD operator =(ThreeD op1, ThreeD op2)
```

Перегрузка оператора + для объектов типа ThreeD

Перегрузка оператора − для объектов типа ThreeD

```
   {
    ThreeD result = new ThreeD();

    /* Обратите внимание на порядок следования операндов — op1 является
       левым операндом, а op2 — правым. */
    result.x = op1.x - op2.x; // Оператор -применяется к значениям,
    result.y = op1.y - op2.y; // имеющим тип int.
    result.z = op1.z - op2.z;

    return result;
   }

  // Вывод значений координат X, Y, Z.
  public void show()
  {
  Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  public static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD();

    Console.Write("Координаты объекта a: ");
    a.show( );
    Console.WriteLine();
    Console.Write("Координаты объекта b: ");
    b.show( );
    Console.WriteLine();

    c = a + b; // Операция сложения объектов a и b.
    Console.Write("Результат операции сложения объектов a и b: ");
    c.show();
    Console.WriteLine();

    c = a + b + c; // Операция сложения объектов a, b и c.
    Console.Write("Результат операции сложения объектов a, b и c: ");
    c.show();
    Console.WriteLine();

    c = c - a;  // Вычитание объекта a из объекта c.
    Console.Write("Результат выполнения операции вычитания c - a: ");
    c.show( ) ;
    Console.WriteLine();

    c = c — b; // Вычитание объекта b из объекта c.
    Console.Write("Результат выполнения операции вычитания c - b: ");
    c.show( );
    Console.WriteLine();
  }
}
```

Применение в выражениях объектов класса `ThreeD`, а также операторов + и –.

Результат выполнения программы:

```
Координаты объекта a: 1, 2, 3

Координаты объекта b: 10, 10, 10

Результат операции сложения объектов a и b: 11, 12, 13

Результат операции сложения объектов a, b и c: 22, 24, 26

Результат выполнения операции вычитания c — a: 21, 22, 23

Результат выполнения операции вычитания c - b: 11, 12, 13
```

Проанализируем текст программы с момента перегрузки оператора +. К двум объектам типа ThreeD применяется оператор +, в результате чего значения соответствующих координат складываются так, как определено в методе operator +(). Однако использование данного метода не приводит к обновлению значений переменной каждого операнда. После выполнения метода возвращается новый объект типа ThreeD, содержащий результат выполнения операции. Чтобы понять, почему операция сложения не изменяет содержимое объекта, представьте действие стандартной арифметической операции сложения, например, в выражении 10 + 12. Результатом выполнения операции будет число 22, но ни число 10. ни число 12 не изменяются результате ее применения. Хотя не существует правила, запрещающего оператора после перегрузки изменять значения одного из операндов, желательно, чтобы после перегрузки оператор сохранял свое первоначальное предназначение.

Как говорилось ранее, метод operator +() возвращает объект типа ThreeD. Хотя метод может возвращать значение любого допустимого в C# типа, то, что оператор возвращает объект типа ThreeD, позволяет применять этот оператор в выражениях состоящих из нескольких операндов, например, a+b+c. Здесь, в результате вычисления выражения a+b возвращается результат, принадлежащий типу ThreeD. Затем этот результат (объект) прибавляется к объекту c. Если же в результате выполнения выражения a+b будет возвращено значение, принадлежащее иному типу, подобное выражение будет недействительным.

Внутри метода operator +() сложение значений отдельных координат сводится сложению целых чисел. Отдельные координаты x, y и z представлены значениям принадлежащими целочисленному типу, поэтому перегрузка оператора + для объектов класса ThreeD не окажет влияния на действия, производимые этим оператору над с целыми числами.

Рассмотрим метод operator - (). Оператор - функционирует так же, как оператор, но при его использовании важен порядок расположения параметров. Напомним, что сложение коммутативно, а вычитание нет (то есть выражение A - B не тождественно выражению B - A). Для всех бинарных операторов первый параметр в методе operator является левым операндом, второй параметр — правым. При перегрузке некоммутативных операторов необходимо учитывать взаимное расположение операндов.

Перегрузка унарных операторов

Перегрузка унарных операторов реализуется аналогично перегрузке бинарных операторов, но при этом используется только один операнд. В следующем примере рассматривается метод, перегружающий в классе ThreeD оператор унарного минуса.

```
// Перегрузка унарного -.
public static ThreeD operator -( ThreeD op)
```

```
{ ThreeD result = new ThreeD();
  result.x - -op.x
  result.y - -op.y
  result.z - -op.z

  return resu.L;
}
```

К значениям объекта, переданного в качестве параметра методу `operator -()`, применяется оператор унарный минус (то есть положительные значения становятся отрицательными). Полученные значения присваиваются переменным созданного объекта `result`, затем этот объект возвращается оператором `return;` в качестве результата. Обратите внимание, что операнд остался неизменным. Это не противоречит обычному предназначению унарного минуса. Например, в выражении

```
a = -b
```

операнду `a` присваивается значение операнда `b` со знаком минус, но значение операнда `b` при этом не изменяется.

Однако в двух случаях (при использовании операторов инкремента и декремента) в методе `operator` должно быть изменено значение операнда. Поскольку обычным предназначением операторов `++` и `--` является увеличение или уменьшение соответственно значения операнда на единицу, то перегруженные версии названных операторов тоже должны выполнять эти действия. То есть при перегрузке оператора инкремента или декремента операнд должен быть изменен. В качестве примера рассмотрим переопределение метода `operator ++()` для класса `ThreeD`.

```
// Перегрузка унарного оператора ++.
public static ThreeD operator ++(ThreeD op)
{
  // Выполняется изменение аргумента.
  op.x++;
  op.y++;
  op.z++;

return op; // Возвращается измененный операнд.
}
```

Оператор ++ изменяет свой операнд.

Ниже приводится расширенная версия предыдущей программы, демонстрирующая использование унарных операторов – и ++.

```
// В программе демонстрируется использование перегруженных операторов.
using System;

   // Класс, содержащий значения координат объекта в трехмерном пространстве.
  class ThreeD {
      int x, y, z; // Координаты объекта.

  public ThreeD() { x = y = z = 0; }
          public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }
  // Перегрузка бинарного оператора +.


    public static ThreeD operator +(ThreeD op1, ThreeD op2)
   {
     ThreeD result = new ThreeD();
```

```
    /* Сложение значений соответствующих координат двух объектов
       и возврат результата - объекта result. */
    result.x = op1.x + op2.x; // В этих трех строках кода выполняется
    result.y = op1.y + op2.y; // сложение целых чисел, для которых
    result.z = op1.z + op2.z; // оператор + сохраняет свое исходное
                                  // предназначение.

    return result;
  }

  // Перегрузка бинарного оператора —.
  public static ThreeD operator -(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    /* Обратите внимание на порядок следования операторов. Операнд op1
       находится слева, а op2 — справа. */
    result.x = op1.x — op2.x; // Из одного целочисленного значения
    result.y = op1.y - op2.y; // вычитается другое целочисленное
                                  // значение.
    result.z = op1.z — op2.z;

    return result;
  }

  // Перегрузка унарного оператора —.
  public static ThreeD operator —(ThreeD op)
  {
    ThreeD result — new ThreeD();

    result.x = —op.x;
    result.y = —op.y;
    result-z = -op.z;

    return result;
  }

  // Перегрузка унарного оператора ++.
  public static ThreeD operator ++ (ThreeD op)
  {
    // Выполняется изменение аргумента.
    op.x++;
    op.y++;
    op.z++;

    return op;


  // Вывод значений координат X, Y, Z.
  public void show()

    Console.WriteLine(x + ", " + y + ", " + z);
  }

}

class ThreeDDemo {
```

```
  public static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD ();

    Console.Write("Координаты объекта a: ");
    a.show();
    Console.WriteLine();
    Console.Write("Координата объекта b: ");
    b.snow();
    Ccrnsole.WriteLine();

    c = a + b; // Операция сложения объектов a и b.
    Console.Write("Результат операции сложения объектов a и b: ");
    c.show();
    Console.WriteLine();

    c = a + b + c; // Операция сложения объектов a, b и c
    Console.Write CPезультат операции сложения объектов a. b и c: ");
    c.show();
    Console.WriteLine{);

    c = c - a; // Объект a вычитается из объекта b.
    Console-Write("Результат выполнения операции вычитания c - a: ");
    c.show();
    Console.WriteLine();

    c = c — b; // Объект b вычитается из объекта c.
    Console.Write("Результат выполнения операции вычитания c - b: ");
    c.show();
    Console.WriteLine();

    c = -a; // Объекту c присваивается значение объекта a,
         // взятое со знаком минус. (Переменным объекта c присваиваются
         // значения объекта a, взятые со знаком минус.)
    Console.write("Результат выполнения операции —a: ");
    c.show();
    Console.WriteLine();

    a++;  // Выполнение перегруженного оператора++
    Console.Write("Результат выполнения операции a++: ");
    a.show();
   }
 }
```

Результат выполнения программы:

```
  Координаты объекта a: 1, 2, 3

Координаты объекта b: 10, 10, 10

Результат операции сложения объектов a и b: 11, 12, 13

Результат операции сложения объектов a, b и c: 22, 24, 26

  Результат выполнения операции вычитания c  - a: 21, 22, 23

Результат выполнения операции вычитания c — b: 11, 12,  13
```

```
Результат выполнения операции -a: -1, -2, -3

Результат выполнения операции a++: 2, 3, 4
```

Легко заметить, что операторы ++ и -- имеют префиксную и постфиксную форму Например, обе формы записи оператора

```
++0;
```

и

```
0++;
```

являются правильными. Однако в результате перегрузки оператора ++ или -- для обеих форм (постфиксной и префиксной) вызывается один и тот же метод. При выполнении перегрузки невозможно указать разницу между префиксной и постфиксной формами этих операторов.

Минутный практикум

1. Что происходит при перегрузке операторов? Какое ключевое слово при этом используется?
2. Каков синтаксис перегружаемого бинарного оператора?
3. Что придает уникальность перегружаемым операторам ++ и --?

Дополнительные возможности класса `ThreeD`

Метод оператора может быть перегружен для любого класса или оператора. Давайте вернемся к классу `ThreeD`. Мы уже рассматривали перегрузку оператора +, с помощью которого суммировались координаты двух объектов, принадлежащих к классу `ThreeD` Однако это не единственный способ изменения значения каждой координаты объекта класса `ThreeD`. Еще, например, можно использовать метод, в котором к каждому значению координат прибавляется целочисленное значение. Для выполнения подобной операции может понадобиться вторичная перегрузка оператора + (как показано ниже):

```
// Перегрузка бинарного оператора +, первый операнд которого является
// объектом класса ThreeD, а второй имеет тип int.
public static ThreeD operator +(ThreeD op1, int op2)
{
  ThreeD result = new ThreeD();

  result.x = op1.x + op2;
  result.y = op1.y + op2;
  result.z = op.z + op2;

  return result;
}
```

В этом методе переопределяется оператор + для выполнения операции сложения между операндами, один из которых имеет тип ThreeD, а другой принадлежит типу int.

1. При перегрузке для оператора определяются действия, которые он будет выполнять по отношению к создаваемому классу. Используется ключевое слово `operator`.

2. Перегрузка бинарного оператора имеет следующий синтаксис:

```
public static ret-type operator op(param-type1 operand1, param-type operand2)
 (
     //операции
 )
```

3. Когда перегружается оператор ++ или -- методом operator изменяется значение операнда. При перегрузке других операторов значение операнда не изменяется.

Обратите внимание, что второй параметр принадлежит к целочисленному типу. Поэтому в приведенном выше методе допускается сложение значений каждого поля (переменной, определенной в классе) объекта типа ThreeD с целочисленным значением. Эта операция допустима, поскольку при перегрузке бинарного оператора один из операндов должен принадлежать к типу класса, для которого перегружается данный оператор, а второй операнд может принадлежать к любому другому типу.

Ниже приведена версия программы, включающая два метода, которые перегружают оператор + в классе ThreeD.

```
   /* В классе ThreeD перегружается оператор + для пары операндов, имеющих тип
   ThreeD, и для пары операндов, один из которых имеет тип ThreeD,
   а другой - тип int. */
using System;

// Класс, содержащий координаты объекта в трехмерном пространстве.
class ThreeD {
  int x, y, z; // Координаты объекта.

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

  // Перегрузка бинарного оператора + для пары операндов,
  // принадлежащих к типу ThreeD.
  public static ThreeD operator +(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    /* Сложение значений соответствующих координат двух объектов
       и возвращение результата. */
    result.x = op1.x + op2.x; // В этих трех строках кода выполняется
    result.y = op1.y + op2.y; // операция сложения целых чисел, для
                              // которых оператор + сохраняет
    result.z = op1.z + op2.z; // свое исходное предназначение.

    return result;
  }

  // Перегрузка бинарного оператора +, операндами которого в данном
  // случае являются объект класса ThreeD и переменная типа int.
  public static ThreeD operator +(ThreeD op1, int op2)
  {
    ThreeD result = new ThreeD();

    result.x = op1.x + op2;
    result.y = op1.y + op2;
    result.z = op1.z + op2;

    return result;
  }

  // Отображение координат X, Y, Z.
  public void show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
```

```
}

class ThreeDDemo {
  public static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD();

    Console.Write("Координаты объекта a: ");
    a.show();
    Console.WriteLine();
    Console.Write("Координаты объекта b: ");
    b.show();
    Console.WriteLine();

    c = a + b; // Объект + объект.
    Console.Write("Результат выполнения операции сложения " +
                  "объектов a и b:  ");
    c.show();
    Console.WriteLine();

    c = b + 10; // Объект целое число.
    Console.Write("Результат выполнения выражения b + 10: ");
    c.show();
  }
}
```

Результат выполнения программы:

```
Координаты объекта a: 1, 2, 3

Координаты объекта b: 10, 10, 10

Результат выполнения операции сложения объектов a и b: 11, 12, 13

Результат вычисления выражения b + 10: 20, 20, 20
```

Выводимые результаты подтверждают, если оператор + применяется к двум объектам, то их координаты суммируются. Если оператор + применяется к объекту и целочисленному значению, координаты объекта увеличиваются на это целочисленное значение.

Хотя при перегрузке оператора + класс `ThreeD` приобретает некоторые полезные свойства, для нормальной работы оператора требуется его вторичная перегрузка. Причины этого в том, что для метода `operator+(ThreeD, int)` допустимо использование операторов, подобных указанному ниже:

```
ob1 = ob2 + 10;
```

Но для этого метода не допускается такая запись оператора:

```
ob1 = 10 + ob2;
```

Дело в том, что в этом операторе целочисленный аргумент располагается слева от оператора +, а метод `operator+()` ожидает, что он будет находиться справа. Для того чтобы можно было использовать обе формы записи оператора, потребуется перегрузить оператор + еще раз. При этом первый операнд должен принадлежать к типу `int`, а второй — к типу `ThreeD`. То есть должна существовать возможность применения оператора + как для операции *объект+целочисленное значение,* так и для

операции *целочисленное значение+объект*. Таким образом, двойная перегрузка оператора + (или любого другого бинарного оператора) обеспечит использование встроенного типа (в данном случае int) как в левой, так и в правой части оператора. Ниже показана версия класса ThreeD, в которой оператор + перегружается согласно описанному алгоритму:

```
/* перегрузка оператора + для операций "объект + объект",
   "объект целое число" и "целое число + объект". */
using System;

// Класс, содержащий значения координат объекта в трехмерном
// пространстве.
class ThreeD {
  int x, y, z; // Координаты объекты.

  public ThreeD() { x = y = z = 0; }
  public ThreeD (int i, int j, int k) { x = i; y = j; z = k; }

  // Перегрузка бинарного оператора + для операции "объект + объект",
  public static ThreeD operator +(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    /* Сложение значений соответствующих координат двух объектов
       и возврат результата. */
    result.x = op1.x + op2.x; // В этих трех строках кода выполняется
    result.y = op1.y + op2.y; // сложение целых чисел, для которых
    result.z = op1.z + op2.z; // оператор + сохраняет
                              // свое исходное предназначение.

    return result;
  }

  // Перегрузка бинарного оператора + для операции "объект +
  // целочисленное значение".
  public static ThreeD operator +(ThreeD op1, int op2)
  {
    ThreeD result = new ThreeD();

    result.x = op1.x + op2;
    result.y = op1.y + op2;
    result.z = op1.z + op2;

    return result;
  }

  // Перегрузка бинарного оператора + для операции
  // "целочисленное значение - объект".
  public static ThreeD operator + (int op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    result.x = op2.x + op1;
    result.y = op2.y + op1;
    result.z = op2.z + op1;
```

Перегрузка оператора, в результате которой можно выполнять операцию *объект + целочисленное значение*.

Перегрузка оператора, в результате которой можно выполнять операцию *целочисленное значение + объект*.

```
    return result;
  }

  // Отображение координат X, Y, Z.
  public void show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  public static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10,10);
    ThreeD c = new ThreeD();

    Console.Write("Координаты объекта a: ");
    a.show();
    Console.WriteLine();
    Console.Write("Координаты объекта b: ");
    b.show();
    Console.WriteLine();

    c = a + b; // Операция объект + объект.
    Console.Write("Результат выполнения операции a + b: ");
    c.show();
    Console.WriteLine();

    c = b + 10;   // Операция объект + целочисленное значение.
    Console.Write("Результат выполнения операции b + 10: ");
    c.show();
    Console.WriteLine();

    c = 15 + b; // Операция целочисленное значение + объект.
    Console.Write("Результат выполнения операции 15 + b:         ");
    c.show();
  }
}
```

Результат выполнения программы:

```
Координаты объекта a: 1, 2, 3

Координаты объекта b: 10, 10, 10

Результат выполнения операции a + b: 11, 12, 13

Результат выполнения операции b + 10: 20, 20, 20

Результат выполнения операции 15 + b: 25, 25, 25
```

Перегрузка операторов сравнения

Операторы сравнения, такие как == или <, также могут быть перегружены, причем этот процесс является очень простым. Как правило, перегруженные операторы сравнения возвращают значение `true` или `false`. В результате остается возможность обычного использования этих операторов, а перегруженные операторы сравнения могут использоваться в условных выражениях. Если возвращается какой-либо иной

тип результата, то изменяется первоначальное предназначение данного оператора. что нежелательно.

Ниже описана версия класса ThreeD, в которой выполняется перегрузка операторов < и > Согласно перегруженным версиям этих операторов один объект считается больше другого, если все координаты данного объекта больше соответствующих координат второго объекта (и один объект считается меньше другого, если все его координаты меньше координат другого объекта).

```
// В программе демонстрируется использование
// перегруженных операторов "<" и ">".
using System;

// Класс, содержащий значения координат объекта в трехмерном
// пространстве,
class ThreeD {
  int x, y, z; // Координаты объекта

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = i; z = k; }

  // Перегрузка оператора "<".
  public static bool operator <(ThreeD op1, ThreeD op2)
  {
    if((op1.x < op2.x) && (op1.y < op2.y) && (op1.z < op2.z))
      return true;
    else
      return false;
  }

  // Перегрузка оператора ">".
  public static bool operator >(ThreeD op1, ThreeD op2)
  {
    if((op1.x > op2.x) && (op1.y > op2.y) && (op1.z > op2.z))
      return true;
    else
      return false;
  }

  // Отображение координат X, Y, Z.
  public void show()

    Console.WriteLine(x + ", " + y + ", " + z);


class ThreeDDemo (
  public static void Main() {
    ThreeD a = new ThreeD(5, 6, 1);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD(1, 2, 3);

    Console.Write("Координаты объекта a: ");
    a.show();
    Console.Write("Координаты объекта b: ");
    b.show();
    Console.Write("Координаты объекта c: ");
    c.show();
    Console.WriteLine();
```

Возвращает значение true, если объект op1 меньше, чем объект op2.

Возвращает значение true, если объект op1 больше, чем объект op2.

```
    if(a > c) Console.WriteLine("Выражение a > c верно.");
    if(a < c) Console.WriteLine("Выражение a < c верно.");
    if(a > b) Console.WriteLine("Выражение a > b верно.");
    if(a < b) Console.WriteLine("Выражение a < b верно.");
  }
}
```

Результат выполнения программы

```
Координаты объекта a: 5, 6, 7
          Координаты объекта b: 10, 10, 10
Координаты объекта c: 1, 2, 3

Выражение a > c верно.
Выражение a < b верно.
```

Перегрузку операторов сравнения необходимо осуществлять попарно (например, если перегружается оператор <, нужно также перегрузить оператор > и наоборот). В C#, как и в любом другом языке программирования, существуют следующие пары операторов:

```
==      !=

<       >
<=      >=
```

Примечание

Если перегружается пара операторов == и !=, то обычно требуется переопределение методов `Object.Equals()` и `Object.GetHashCode()`. Эти методы и технология их переопределения рассматриваются в главе 8.

Основные положения и ограничения при перегрузке операторов

Действие перегруженного оператора, применимое к объекту класса, для которого он определен, не связано со стандартным использованием оператора для встроенных типов C#. Во избежание усложнения чтения программы по возможности перегружайте операторы так, чтобы они выполняли действия, дли которых изначально предназначались. Например, оператор +, перегруженный для работы, с объектом класса `ThreeD`, подобен оператору +, определенному для целочисленных типов данных. Если по отношению к какому-либо классу определить действия оператора + так, что вместо операции сложении будет выполняться операция деления, это просто приведет к путанице, не давая никаких преимуществ.

Перегрузка операторов связана с определенными ограничениями. Так, невозможно изменить приоритет операторов. Также запрещено изменение количества операндов, в которых нуждается оператор. Кроме того, существует множество операторов, которые не могут быть перегружены. В первую очередь к ним относятся оператор присваивания и составные операторы присваивания, такие как +=. Ниже приводится перечень других операторов, которые не могут быть перегружены.

```
  &&      ||       []       ()          new       is
 silect  Typeof    ?        ->           .        =
```

Минутный практикум

1. Что требуется сделать, чтобы для переопределенного оператора / была возможность указывать первым операндом переменную типа `int`, а вторым — объект необходимого класса и наоборот?
2. Какой тип данных обычно возвращают перегруженные операторы сравнения?

3. Можно ли перегружать оператор присваивания?

Ответы профессионала

Вопрос. Вы говорите, что составные операторы присваивания перегружать нельзя. Что произойдет, если в качестве операнда для составного оператора присваивания += указать объект класса, для которого был определен оператор +?

Ответ. В общем случае, если какой-либо оператор был переопределен, то при его использовании в составном операторе присваивания вызывается метод перегруженного оператора. Таким образом, оператор += автоматически использует пользовательскую версию метода `operator +()`. Например, если при работе с классом `ThreeD` воспользоваться строками кода

```
ThreeD a = new ThreeD(1, 2, 3);
ThreeD b = new ThreeD(10, 10, 10);

b == a; //Выполняется операция сложения объектов a и b
```

происходит автоматический вызов метода `operator +()` для объекта типа `ThreeD`, причем при его выполнении переменной `b` будут присвоены координаты 11, 12, 13.

Индексаторы

Вы знаете, что индексация массивов производится с помощью оператора `[]`. Этот оператор может быть перегружен для создаваемых вами классов, но в таком случае не будет использоваться метод `operator`. Вместо этого метода создается *индексатор*. В результате к содержащемуся в объекте массиву можно осуществить доступ, указав имя объекта, а не имя массива. Основная область применения индексаторов — поддержка создания специализированных массивов, на которые накладывается одно или несколько ограничений. Индексаторы позволяют работать с любыми значениями, используя почти тот же синтаксис, что и при работе с массивами. Индексаторы могут иметь одно или несколько измерений.

Начнем рассмотрение темы с одномерных индексаторов, которые определяются с использованием следующего синтаксиса:

```
clement-type this[int index] {
 // Метод доступа get.
 get {
    // Возврат значения, указанного с помощью параметра index.
```

---

1. Оператор / требуется перегружать двумя методами, причем в одном случае объект должен быть вторым параметром, в другом случае — первым.

2. Операторы сравнения обычно возвращают значение типа `bool`.

3. Нет.

```
  }

  // Метод доступа set
  set {
    // Присваивание (например, элементу массива) значения,
    // указанного с помощью параметра index.
  }
}
```

Здесь вместо словосочетания `element-type` указывается базовый тип индексатора. В результате каждый элемент, доступ к которому осуществляется с помощью индексатора. должен принадлежать к типу `element-type` (то есть будет выполняться доступ к массиву, элементы которого имеют указанный тип). Вместо параметра `index` указывается индекс элемента, к которому производился доступ. Этот параметр не обязательно должен принадлежать к типу `int`, но поскольку индексаторы обычно применяются для индексирования элементов массива, чаще всего используется именно целочисленный тип.

В составе индексатора определены два метода доступа, `get` и `set`. Структура метода доступа подобна структуре обычного метода за исключением того, что для метода доступа не указывается тип возвращаемого значения и он не имеет параметров. Оба метода доступа при использовании индексатора вызываются автоматически и принимают в качестве параметра значение, указанное вместо слова `index`. Если индексатор находится в левой части оператора присваивания, вызывается метод доступа `set`, а значение второго операнда присваивается элементу, указанному с помощью параметра `index`. Значение передается методу доступа с помощью неявного параметра `value` (в приводимой далее программе обратите внимание на синтаксис применения этого неявного параметра — переменная `value` нигде не объявляется, и ее тип нигде не указывается). Если индексатор находится в правой части оператора присваивания, то вызывается метод доступа `get`, в результате выполнения которого возвращается значение, ассоциированное с параметром `index`.

Одно из преимуществ, обеспечиваемых индексатором, заключается в том, что он гарантирует возможность точного контроля над доступом к массиву (предотвращается некорректный доступ). Например, использование индексаторов является наилучшим вариантом реализации массива с применением алгоритма предотвращения сбоев, созданного в главе 6. (При этом в код программы внедряется условный оператор `if`, проверяющий, не выходит ли указываемый индекс за пределы массива, то есть просто не допускается возникновение ошибочной ситуации.) Это достигается путем применения индексатора, обеспечивающего доступ к массиву с помощью обычного синтаксиса, используемого при работе с массивами.

```
// Для доступа к элементам массива в программе применяется индексатор,
// в котором используется алгоритм предотвращения сбоев.
using System;

class FailSoftArray {
  int [] a; // Ссылка на массив.

  public int Length; // Переменная Length объявляется как public.

  public bool errflag; // Переменная возвращает значение, указывающее,
       // выходит индекс, передаваемый индексатору в качестве
       // параметра, за пределы массива или нет.
```

```
  // Создание массива заданного размера,
  public FailSoftArray (int. size) {
    a = new int[sizel;
    Length = size;
  }

   // Индексатор для класса FailSoEtArray.
  public int this[int index] {
    // Метод доступа get.
    get {
      if (ok(index))  {
        errflag — false;
        return a[index];
      } else {
        errflag = true;
        return 0;
      }
    }

  // Метод доступа set.
    set {
      if (ok(index))  {
        a[index] = value;
        errflag = false;
      }
      else errflag = true;
    }
  }

  // Возвращает значение true, если индекс находится в заданных
  // границах массива,
  private bool ok(int index) {
    if(index >= 0 & index < Length) return true;
    return false;
  }


// Показана работа с массивом с применением алгоритма предотвращения сбоев.
class ImprovedFSDemo {
  public static void Main()     {
    FailSoftArray fs = new FailSoftArray(5);
    int x;

    // При указании индекса, выходящего за пределы массива,
    // выводится соответствующее сообщение.
    Console.WriteLine("Применение алгоритма предотвращения сбоев.");
    for(int i=0; i < (fs.Length * 2); i++)
      fs[i] = i * 10;

    for(int i=0; i < (fs.Length * 2); i++) {
      x - fs[i] ;
      if(x != -1) Console.Write(x + " ");
    }
    Console.WriteLine();

    // Задается условие, приводящее к ошибке.
```

Индексатор для класса `FailSoftArray`.

Вызов метода доступа `set`.

Вызов метода доступа `get`.

```
    Console.WriteLine("\nСбой с сообщением об ошибке.");
    lor(int i=0; i < (fs.Length * 2); 1++) {
      fs[i] = i*10;
      if(fs.errflag)
        Console.WriteLine("При обращении к элементу fs[" + i +
                    "] превышены граничные значения индекса массива.");
    }

    for(int i=0; i < (fs.Length * 2); i++) {
      x = fs[i];
      if (!fs.errflag) Console.Write (x + " ");
      else
        Console.Write("\nПри обращении к элементу fs[" + i +
                    "] превышены граничные значения индекса массива.");

  }
 }
}
```

Результат выполнения программы:

```
Применение алгоритма предотвращения сбоев.
0 10 20 30 40 0 0 0 0 0

Сбой с сообщением об ошибке.
При обращении к элементу fs[5] превышены граничные значения индекса массива
При обращении к элементу fs[6] превышены граничные значения индекса массива
При обращении к элементу fs[7] превышены граничные значения индекса массива
При обращении к элементу fs[8] превышены граничные значения индекса массива
При обращении к элементу fs[9] превышены граничные значения индекса массива
0 10 20 30 40
При обращении к элементу fs[5] превышены граничные значения индекса массива
При обращении к элементу fs[6] превышены граничные значения индекса массива
При обращении к элементу fs[7] превышены граничные значения индекса массива
При обращении к элементу fs[8] превышены граничные значения индекса массива
При обращении к элементу fs[9] превышены граничные значения индекса массива
```

Этот результат аналогичен результату выполнения программы из главы 6. В данной версии используется индексатор, предотвращающий превышение граничных значении индекса массива. Теперь подробнее рассмотрим организацию индексатора. Объявление индексатора начинается со следующей строки:

```
public int this[int index] {
```

Здесь объявляется индексатор, который производит операции с элементами, имеющими тип `int`. Индексатор объявляется как `public` и может использоваться кодом, не относящимся к его классу. Ниже показан код метода доступа `get`:

```
get {
  if (ok(index)) {
    errflag = false;
    return a[mcex];
  } else {
    errflag = true;
    return 0;
  }
}
```

Благодаря использованию метода доступа `get` предотвращаются ошибки, возникающие при превышении граничных значений индекса массива. Если указанный индекс

находится внутри границ, возвращается элемент, соответствующий этому индексу. Если индекс выходит за пределы массива, то выполняется строка кода `return 0;`, то есть вместо элемента массива возвращается значение `0`. В данной версии класса `FailSoftArray` переменная `errflag` содержит результат проверки (то есть проверяется, выходит ли указываемый индекс за пределы массива). Это поле (переменная) может проверяться после выполнения каждой попытки доступа для оценки ее успешности и для выведения информации об ошибке. (В главе 10 будет рассмотрен еще один способ обработки ошибок с помощью подсистемы исключений C#, но на данном этапе достаточно использования флага ошибки, то есть булевой переменной `errflag`.)

Ниже приводится код метода доступа `set`, в котором также предотвращается превышение граничных значений индекса массива.

```
set {
  if(ok(index))   {
    a[index] = value
    errflag = false;
  }
  else errflag = true;
}
```

Если параметр `index` не выходит за пределы массива, значение, передаваемое неявным параметром `value`, присваивается соответствующему элементу. В противном случае параметру `errflag` присваивается значение `true`. Как вы помните, в методе доступа используется неявный параметр `value`, который содержит присваиваемое значение. Этот параметр не требует объявления.

Методы доступа `get` и `set` не должны поддерживаться индексатором одновременно. Можно создать индексатор, предназначенный лишь для извлечения элемента, выполнив реализацию только метода доступа `get`. Можно создать индексатор, предназначенный лишь для присваивания значения элементу массива, реализовав только метод доступа `set`.

Индексатор не обязательно должен работать с массивом. Просто для тех, кто использует индексаторы, способ работы с элементами индексатора должен быть похож на способ работы с массивом. Например, в следующей программе используется индексатор, подобный массиву, который содержит степени числа 2 (от нулевой до пятнадцатой) и предназначен только для чтения. Однако обратите внимание на то, что массива в данном случае не существует. Вместо этого индексатор просто вычисляет подходящее значение, исходя из указываемого индекса.

```
// Индексаторы не обязательно должны работать с массивами.
using System;

class PwrOfTwo {

  /* Доступ к логическому массиву, который содержит степени числа 2
     (от нулевой до пятнадцатой).     */
  public int this[int index] {
    // Вычисление и возврат степени числа 2.
    get {
      if ((index >= 0) && (index < 16)) return pwr(index);
      else return -1;
    }
```

```
    // Отсутствует метод доступа set.
  }

  int pwr (int p) {
    int result = 1;

    for(int i=0; i<p; i++)
      result *= 2;

    return result;
  }
}

class UsePwrOfTwo {
  public static void Main() {
    PwrOfTwo pwr = new PwrOfTwo();

    Console.Write("Первые восемь степеней числа 2: ");
    for(int i=0; i < 8; i++)
      Console.Write(pwr[i] + " ");
    Console.WriteLine();

    Console.Write("При указании индексов -1 и 17 возвращены "+
                  "результаты:   ");
    Console.Write(pwr [-11 + " " + pwr[17]);
  }
}
```

Результат выполнения программы:

```
Первые восемь степеней числа 2: 1 2 4 8 16 32 64 128
При указании индексов -1 и 17 возвращены результаты: -1 -1
```

Заметьте, что в индексаторе для класса `PwrOfTwo` присутствует метод доступа `get`, но отсутствует метод доступа `set`. Это означает, что индексатор предназначен только для чтения. Следовательно, объект класса `PwrOfTwo` может находиться в правой части оператора присваивания, но не может находиться в левой. Например, будет неудачной попытка использования в предыдущей программе оператора

```
pwr[0] — 11; // Этот оператор не может быть скомпилирован.
```

Использование этого оператора приведет к ошибке компиляции, потому что для индексатора не определен метод доступа `set`.

При использовании индексаторов существует одно важное ограничение: для них не определена возможность применения параметра `ref` или `out`.

Многомерные индексаторы

Индексаторы могут создаваться и для многомерных массивов. Рассмотрим двухмерный массив (и двухмерный индексатор), в котором используется алгоритм предотвращения сбоев. Обратите внимание на способ объявления индексатора.

```
//В программе демонстрируется применение двухмерного индексатора,
// использующего алгоритм предотвращения сбоев,
using System;

class FailSoftArray2D {
  int[,] a; // Ссылка на двухмерный массив.
```

```
    int rows, cols; // Ряды и колонки двухмерного массива.
  public int Length; // Переменная Length объявляется открытой.

    public bool errflag; // В переменной errflag фиксируется результат
                         // попытки доступа к элементу массива.

  // Создание массива по заданным размерам.
  public FailSoftArray2D(int r, int c) {
    rows = r;
    cols = c;
    a = new int[rows, cols];
    Length = rows * cols;
  }

  // Индексатор для класса FailSoftArray2D.
  public int this[int index1, int index2] {
    // Метод доступа get.
    get {
      if (ok(index1, index2)) {
        errflag = false;
        return a[index1, index2];
      } else {
        errflag = true;
        return 0;
      }
    }

    // Метод доступа set.
    set {
      if(ok(index1, index2)) {
        a[index1, index2] = value;
        errflag = false;
      }
      else errflag = true;
    }
  }

  // Возвращение значения true,
  // если индексы не выходят за граничные значения индексов массива,
  private bool ok(int index1, int index2) {
    if(indexl >= 0 & index1 < rows &
      index2 >= 0 & index2 < cols)
        return true;

    return false;
  }
}

// Использование двухмерного индексатора.
class TwoDIndexerDemo {
  public static void Main() {
    FailSoftArray2D fs = new FailSoftArray2D(3, 5);
    int x;

    // При выполнении следующего цикла for происходит обращение к
    // несуществующим элементам массива, то есть возможно возникновение
    // ошибочной ситуации. Однако такая ситуация была обработана ранее —
    // при ее возникновении переменной errflag присваивается значение true и
    // возвращается значение 0.
    Console.WriteLine("При выходе за границы массива возвращается значение 0.");
```

Двухмерный индексатор.

```
    for (int i=0; i < 6; i++)
      fs[i, i] = i*10;

    for(int i=0; i < 6; i++) {
      x = fs[i,i];
      if(x != -1) Console.Write(x + " ");
    }
    Console.WriteLine();

    // Теперь переменная errflag используется как флаг ошибки, то есть
    // является индикатором необходимости вывода сообщения.
    Console.WriteLine("/nВыход за границы массива с сообщением об "+
                      " ошибке.")
    for (int i=0; i < 6; i++) {
      fs[i,i] = i*10;
      if(fs.errflag)
        Console.WriteLine("/nПри обращении к элементу fs[" + i + ",
                  " + i + "] превышены граничные значения индексов массива.");
    }

    for(int i=0; i < 6; i++) {
      x = fs[i,i];
      if(!fs.errflag) Console.Write(x + " ");
      else
        Console.WriteLine("\nПри обращении к элементу fs[" + i + ",
                  " + i + "] превышены граничные значения индексов массива.");
    }
  }
}
```

Результат выполнения программы:

```
При выходе за границы массива возвращается значение 0.
0 10 20 0 0 0

Выход за границы массива с сообщением об ошибке.
При обращении к элементу fs[3, 3] превышены граничные значения индексов массива.
При обращении к элементу fs[4, 4] превышены граничные значения индексов массива.
При обращении к элементу fs[5, 5] превышены граничные значения индексов массива.
0 10 20
При обращении к элементу fs[3, 3] превышены граничные значения индексов массива.
При обращении к элементу fs[4, 4] превышены граничные значения индексов массива.
При обращении к элементу fs[5, 5] превышены граничные значения индексов массива.
```

Минутный практикум

1. Как называются методы доступа в индексаторе, и какие функции они выполняют?

2. Можно ли определить индексатор, как предназначающийся только для чтения?

3. Можно ли создать индексатор для двухмерного массива?

1. Методы доступа называются get и set. Первый считывает значение указанного элемента массива, второй присваивает ему значение соответствующего типа.

2. Да.

3. Да.

Ответы профессионала

Вопрос. Могут ли быть перегружены индексаторы?

Ответ. Да. Вызывается та версия, которая имеет наибольшее соответствие типов между его параметром и аргументом (или аргументами), используемым в качестве индекса.

Свойства

Еще одним специальным типом членов класса, имеющим сходство с индексаторами, является *свойство,* в котором поле связано с методом, обеспечивающим доступ к полю. В некоторых предыдущих программах возникала необходимость создания поли, хотя и доступного из других объектов, но такого, для которого выполнялся бы контроль над операциями с его значением (контроль того, какие операции с его значением возможны, а какие нет). Например, может потребоваться ограничение диапазона значений, присваиваемых этому полю. Конечно, для достижения подобной цели можно использовать закрытую переменную в сочетании с методом, обеспечивающим доступ к ее значению, но проще и лучше во всех отношениях использовать свойства.

Свойство состоит из имени и методов доступа `get` и `set`. Методы доступа используются для получения значения и присваивания его переменной. Основным достоинством свойства является то, что его имя может применяться в выражениях и операциях присваивания подобно обычной переменной, хотя на самом деле при этом автоматически вызываются методы доступа `get` и `set`. Такое использование свойства похоже на автоматический вызов методов `get` и `set` в индексаторе.

Синтаксис свойства выглядит следующим образом:

```
type name {
  get {
    // Код метода доступа get.
  }

  set {
    // Код метода доступа set.
  }
}
```

Здесь параметр `type` указывает тип свойства (например, тип `int`), а параметр `name` — имя свойства. После определения свойства любое использование идентификатора, указанного вместо `name`, приводит к вызову соответствующего метода доступа. Метод доступа `set` автоматически получает параметр `value`, содержащий значение, которое присваивается свойству.

Свойство управляет доступом к полю. Однако в свойстве поле не объявляется, поскольку оно должно быть создано вне свойства.

В приводимой ниже простой программе определяется свойство тургор, которое обеспечивает доступ к полю `prop`. В данном случае свойство допускает присваивание полю только положительных значений.

```
// В программе демонстрируется использование простого свойства.
 using System;
```

```
class SimpProp {
  int prop; // Поле, доступ к которому осуществляется с помощью метода
            // myprop, в котором также определяется, какие именно
            // значения могут быть присвоены данному полю.

  public SimpProp() { prop = 0; }

  /* Свойство, обеспечивающее доступ к закрытой
     переменной экземпляра prop. Оно позволяет присваивать полю
     только положительные значения. */
  public int myprop {
    get {
      return prop;
    }
    set }
      if(value >= 0) prop = value;
    }
  }
}

// Использование свойства.
class propertyDemo {
  public static void Main() {
    SimpProp ob = new SimpProp();

    Console.WriteLine("Первоначальное значение свойства ob.myprop: " +
    ob.myprop);

    ob.myprop = 100; //Свойству присваивается значение.
    Console.WriteLine("Значение свойства ob.myprop: " + ob.myprop);

    // Полю prop невозможно присвоить отрицательное значение.
    Console.WriteLine{"Попытка присвоить свойству ob.myprop " +
                      "значение -10.");
    ob.myprop = -10;
    Console.WriteLine("Значение переменной ob.myprop: " ■+ ob.myprop);
  }
}
```

Свойство myprop управляет доступом к полю prop.

Свойство myprop используется подобно переменной.

Результат выполнения программы:

```
Первоначальное значение свойства ob.myprop: 0
Значение свойства ob.myprop: 100
Попытка присвоить свойству ob.myprop значение -10.
Значение переменной ob.myprop: 100
```

Теперь тщательно проанализируем эту программу. В ней определяется одно закрытое поле `prop` и свойство `myprop`, управляющее доступом к этому полю (в свойстве не объявляется поле, а только происходит управление доступом к нему). Таким образом, свойство неразрывно связано с полем, объявленным в классе, где определено это свойство. Более того, поскольку поле `prop` является закрытым, доступ к нему может осуществляться только с помощью свойства тургор.

Свойство `myprop` определяется как `public`, поэтому доступ к нему может осуществляться из других объектов. Такой подход является правильным, поскольку свойство обеспечивает доступ к полю `prop`, которое является закрытым. Метод доступа `get` просто возвращает значение поля, а метод доступа `set` присваивает переменной `prop`

значение только в том случае, если оно является положительным. Таким образом, свойство myprop контролирует значения, которые можно присвоить полю prop.

Свойство myprop является свойством «чтение-запись», поскольку в нем определена возможность и получения, и присвоения значения. Кроме того, можно создавать свойства, так же как индексаторы, предназначенные только для чтения или только для записи. Для создания свойства, предназначенного лишь для чтения, нужно определить только метод доступа get. Для определения свойства, функционирующего исключительно в режиме записи, нужно определить только метод доступа set.

Свойства можно использовать для улучшения программы, в которой обрабатывалась возможность возникновения ошибочных ситуаций. Ранее уже говорилось о том, что все массивы ассоциированы со свойством Length. При этом в классе FailSoftArray использовалось открытое целочисленное поле Length. На самом деле это решение не является оптимальным, поскольку оно позволяет присваивать полю Length некоторое значение, отличное от длины массива. (Например, такое значение может быть преднамеренно искажено.) Эту проблему можно решить путем преобразования свойства Length в свойство, предназначенное только для чтения, как это показано в следующей программе.

```
// В программе демонстрируется использование свойства Length,
using System;

class FailSoftArray {
  int[] a; // Ссылка на массив,
  int len; // Длина массива.

  public bool errflag; // В переменной errflag фиксируется результат
                       // попытки доступа к элементу массива.

  // Конструирование массива заданного размера,
  public FailSoftArray(int size) {
    a = new int[size];
    len = size;
  }

  // Свойство Length, предназначенное только для чтения.
  public int Length {
    get {
      return len;
    }
  }

  // Индексатор для FailSoftArray.
  public int this[int index] {
    // Метод доступа get.
    get {
      if(ok(index)) {
        errflag = false;
        return a[index];
      } else {
        errflag = true;
        return 0;
      }
    }
```

Теперь Length является свойством, а не полем.

```
    // Метод доступа set.
    set {
      if(ok(index))   {
        a[index] = value;
        errflag = false;
      }
      else errflag = true;
    }
  }

  // Возвращает значение true, если индекс не выходит за границы массива.
  private bool ok(int index) {
    if(index >= 0 & index < Length) return true;
    return false;
  }
}

// Использование свойства Length.
class ImprovedFSDemo {
  public static void Main() {
    FailSoftArray fs = new FailSoftArray(5);
    int x;

    // Можно считывать значение свойства Length.
    for(int i=0; i < (fs.Length); i++)
      fs[i] = i*10;

    for(int i=0; i < (fs.Length); i++) {
      x = fs[i];
      if (x != -1) Console.Write(x + " ");
    }
    Console.WriteLine();

    // fs.Length — 10; // Не являясь комментарием, этот оператор будет
    // недействителен, поскольку для свойства не определен метод
    // доступа set.

  }
}
```

Итак, теперь `Length` является свойством, предназначенным только для чтения. Вы можете проверить его недоступность для изменения. Чтобы сделать это, удалите символ комментария, предшествующий оператору

```
// fs.Length = 10;
```

После попытки скомпилировать данный код программа выведет сообщение о том, что свойство `Length` предназначено только для чтения.

Ограничения в использовании свойств

Свойствам присущи некоторые ограничения, на которые следует обратить особое внимание. Во-первых, свойство не может быть передано методу в качестве параметра `ref` или `out`. Во-вторых, оно не может быть перегружено. (Вы можете определить два свойства, имеющих доступ к одной и той же переменной, но лучше не пользоваться этой возможностью.) Наконец, в-третьих, свойство не должно изменять

значение базовой переменной, доступ к которой контролируется свойством, при вызове метода доступа `get`. (Хотя создание такого метода доступа не запрещено компилятором, метод `get` должен соответствовать своему первоначальному предназначению — только считывать данные.)

Минутный практикум

1. Можно ли объявить поле в свойстве?
2. Какие преимущества дает использование свойства?
3. Можно ли передать свойство методу в качестве параметра `ref`?

Проект 7-1. Создание класса `Set`

`SetDevo.cs`

Как уже говорилось, перегрузка операторов, а также использование индексаторов и свойств облегчают создание классов, полностью интегрированных в среду программирования языка C#. Обратите внимание, что с помощью переопределения требуемых операторов, индексаторов или свойств обеспечивается использование в программе объекта, для класса которого и было выполнено переопределение, практически таким же образом, как используется встроенный тип. Работа с объектами такого класса выполняется при помощи операторов и индексаторов, причем эти объекты могут использоваться в выражениях. Включение свойств в класс обеспечивает интерфейс, совместимый со встроенными объектами C#. Данный проект иллюстрирует создание нового класса и его интеграцию в среду C#. В проекте создастся класс `Set`, имитирующий множество, а также для этого класса перегружаются операторы, позволяющие выполнять операции над множествами (объединение, разность, добавление элемента и его удаление).

Исходя из целей данного проекта определим *множество* как коллекцию уникальных элементов. Уникальность означает, что никакие два элемента множества не могут совпадать. Порядок следования элементов множества не имеет значения. Таким образом, множество

```
{A, B, C}
```

идентично множеству

```
{A, C, B}
```

Множество также может быть пустым.

Для множества могут определяться операции. В нашем проекте остановимся на выборе следующих операций:

- добавление элемента в множество;
- удаление элемента из множества;
- объединение множеств;
- разность множеств.

---

1. Нет, переменная должна быть объявлена в классе, в котором и определено свойство.

2. Использование свойства позволяет применять для вызова метода доступа тот же синтаксис, что и при обращении к переменной.

3. Нет.

Добавление и удаление элементов множества не требуют особых объяснений, поэтому подробно рассмотрим оставшиеся две операции.

Объединением двух множеств называется множество, которое включает все элементы обоих множеств (конечно, в этом случае не допускается дублирование элементов). Для объединения множеств в проекте используется оператор +.

Разностью множеств называют множество, содержащее элементы первого множества, которые не входят в состав второго множества. Для вычисления разности двух множеств в проекте используется оператор Например, если рассматривать разность двух множеств S1 и S1, то этот оператор удаляет элементы множества S2 из множества S1, помещая результат в множество S3:

```
S3 = S1 - S2
```

Если множества S1 и S2 совпадают, множество S3 будет пустым.

Конечно, для множеств могут быть определены и другие операции. Некоторые из них рассматриваются в разделе «Контрольные вопросы». Вы можете попытаться самостоятельно определить операции, которые будут полезны для работы с множествами.

С целью упрощения проекта класс `set` будет состоять из наборов символов, но принципы, используемые при определении данного класса, могут применяться и при создании класса, предназначенного для хранения элементов других типов.

Пошаговая инструкция

1. Создайте новый файл с именем `SetDemo.cs`.
2. Начните создание класса `Set` следующим образом:

```
class Set {
char [] members; // Этот массив содержит символы.
int len; // Количество элементов массива.
```

   К каждому из символов, содержащихся в массиве, можно обратиться, указав имя массива и номер этого элемента. Число, указывающее количество элементов массива, хранится в переменной `len`.

3. Добавьте в определение класса `Set` следующие конструкторы:

```
// Создание пустого множества.
public Set()  {
  len = 0;
}

// Создание пустого множества заданного размера.
public Set(int size) {
  members = new char[size]; // Выделение памяти для множества.
  len = 0;   // Множество еще пустое.
}

// Создание множества на базе другого множества (объекта класса Set).
public Set(Set s) {
  members = new char[s.len]; // Выделение памяти для множества.
  for(int i=0; i<s.len; i++) members[i] = s[i];
  len = s.len; // Количество элементов.
}
```

Для создания множества определены три конструктора. Используя первый, можно создать пустое множество. В нем не создастся массив, содержащий элементы множества, а только инициализируется переменная `len`. При использовании второго конструктора создается пустое множество заданного размера. И наконец, множество можно создать на основе другого множества. В этом случае два множества содержат одни и те же элементы, но являются различными объектами.

4. Добавьте свойство `Length` и индексатор, предназначенные только для чтения.

```
// Определение свойства Length, предназначенного только для чтения,
 public int Length {
  get {
     return len;
  }
 }

 // Определение индексатора, предназначенного только для чтения.
public char this[int idx] {
  get {
     if(idx >= 0 & idx < len) return members[idx];
     else return (char)0;
  }
}
```

Свойство `Length` используется для указания шины массива. Индексатор возвращает элемент множества по указанному индексу. Проверка границ массива выполняется с целью предотвращения превышения граничных значений массива. Если индекс некорректен, возвращается символ 0.

5. Добавьте метод `find()`, код которого приведен ниже. Этот метод определяет, содержит ли множество символ, переданный ему с помощью параметра `ch`. Если этот элемент найден, метод возвращает индекс элемента, если нет — значение -1.

```
/*Если элемент входит в состав множества, метод find возвращает индекс элемента, если
    не входит — возвращается значение -1, */
int find(char ch) {
  int i;

  for (i=0; i<len; i++)
    if(members[i] == ch) return i;

  return -1;
}
```

Для выполнения операции добавления элемента в множество перегрузите оператор + для объектов класса `Set`, как это показано ниже.

```
// Добавление уникального элемента в множество
public static Set operator + (Set ob, char ch) {
  Set newset = new Set(ob.len +1); // Увеличение числа элементов
                                        // нового множества на единицу.

  // Копирование элементов.
  for(int 1=0; i < ob.len; i++)
    newset.members[i] = ob.members[i];

  // Переменной len возвращаемого объекта присваивается значение
  // переменной len копируемого объекта, то есть передается значение
  // размера множества,
  newset.len = ob.len;
```

```
  // Проверяется наличие в множестве добавляемого символа.
  if(ob.find(ch) == —1) { // Если элемент не найден,
           // он добавляется в новое множество.
  newset.members[newset.len] = ch;
    newset.len++;
  }
  return newset; // Возврат нового множества (объекта класса Set).
}
```

Рассмотрим работу перегруженного оператора подробнее. Во-первых, создаваемое новое множество содержит элементы исходного множества, которое передается с помощью параметра `ob`, и новый элемент, который передается параметром `ch`. Обратите внимание, что новое множество содержит на один элемент больше, чем первоначальное множество `ob`, то есть добавление нового элемента учтено. Затем исходные элементы копируются в новое множество, а переменная `len`, содержащая значение размера нового множества, становится равной переменной `len` исходного множества. Добавление элемента `ch` в новое множество происходит только в том случае, если этот элемент еще не содержится в данном множестве (то есть вначале вызывается метод `find()`). При добавлении каждого нового элемента происходит приращение значения переменной `len` на единицу. После выполнения этой операции возвращается новое множество — объект `newest`, причем исходное множество не изменяется.

7. Выполните перегрузку оператора −, в ходе выполнения которого должен удаляться элемент множества. Код перегруженного оператора приведен ниже.

```
// Удаление элемента из множества.
public static Set operator - (Set ob, char ch) {
  Set newset = new Set();
  int i = ob.find(ch); // Если элемент не найден,
                       // переменной i Судет присвоено значение -1.

  // Копирование оставшихся элементов с использованием перегруженного
  // оператора +.
  for(int j=0; j < ob.len; j++)
    if(j i= i) newset = newset + ob.members[j];

  return newset;
}
```

Вначале будет создано новое пустое множество, затем вызван метод `find()`, позволяющий определить, входит ли символ, передаваемый параметром `ch`, в состав исходного множества. Как вы помните, метод `find()` возвращает значение -1, если символ, передаваемый параметром `ch`, не входит в состав множества. Затем все элементы исходного множества добавляются в новое множество, за исключением элемента, индекс которого равен значению, возвращаемому методом `find()`. Таким образом, результирующее множество содержит все элементы исходного множества, исключая символ, переданный параметром `ch`. Если символ `ch` не является частью исходного множества, то возвращенное множество будет идентично исходному.

8. Теперь для выполнения операций объединения и разности множеств вновь перегрузите операторы + и −, как показано ниже.

```
// Объединение множеств.
public static Set operator + (Set ob1, Set ob2) {
  Set newset = new Set (ob1); // Копирование первого множества.
```

```
    // Добавление уникальных элементов из второго множества,
    for (int i=0; i < ob2.1en; i++)
      newset — newset + ob2[i];

    return newset; // Возврат нового множества.
  }

  // Разность множеств.
  Public static Set operator -(Set obi. Set ob2) {
    Set newset = new Set(obi); // Копирование первого множества.

    // Вычитание из первого множества элементов второго множества.
    for(int i=0; i < ob2.1en; i++)
      newset = newset - ob2[i];

    return newset; // Возврат нового множества.
  }
```

Как видите, для упрощения выполнения соответствующих операций эти методы используют ранее перегруженные версии операторов + и −. При объединении множеств новое созданное множество вначале содержит элементы из первого множества, затем к ним добавляются элементы второго множества. Поскольку операция + выполняет добавление элемента в множество только в том случае, если он не является частью этого множества, результирующее множество представляет собой объединение двух множеств (без дублирования элементов). Оператор - удаляет соответствующие элементы.

9. Ниже приводится полный код класса `Set` и использование классом `SetDemo` перегруженных операторов и объектов, имитирующих множество.

```
    /* Проект 7.1

В программе создается класс Set, имитирующий множество и выполняющий
    операции над множествами. */

using System;

  class Set {
    char[] members; // Массив, содержащий символы (имитирующий множество)
    int len; // Количество элементов множества.

    // Создание пустого множества,
    public Set()   {
      len = 0;
    }

    // Создание пустого множества заданного размера,
    public Set(int size) {
      members = new char[size]; // Распределение памяти для множества
      len = 0;  // Элементы не были сконструированы
    }

    // Конструирование множества на базе другого множества.
    public Set(Set s) {
      members = new char[s.len]; // Выделение памяти для множества.
```

```
   for(int i = 0; i < s.len; i++) members [i] = s[i];
   len = s.len; // Количество элементов множества.
}

// Определение свойства Length, предназначенного только для чтения.
public int Length {
   get {
      return len;
   }
}
// Определение индексатора, предназначенного только для чтения,
public char this[int idx] {
   get {
      if(idx >= 0 & idx < len) return members[idx];
      else return (char)0;
   }
}

/* Если элемент входит в состав множества, метод find возвращает индекс
 элемента, если не входит, то возвращается значение -1, */
int find(char ch) {
   int i;

   for(i=0; i < len; i++)
      if(members[i] == ch) return i;

   return —1;
}

// Добавление в множество уникального элемента.
public static Set operator +(Set ob, char ch) {
   Set newset = new Set(ob.len + 1); // Увеличение числа элементов
                          // нового множества на единицу.

   // Копирование элементов,
   for(int i=0; i < ob.len; i++)
      newset.members[i] = ob.members[i];

   // Переменной len возвращаемого объекта присваивается значение
   // переменной len копируемого о&ьекта, то есть передается значение
   // размера множества,
   newset.len = ob.len;

   // Выполняется проверка наличия в множестве добавляемого символа.
   if(ob.find(ch) == -1) ( // Если элемент не найден,
             // он добавляется в новое множество,
      newset.members[newset.len] = ch;
      newset.len++;
   }
   return newset; // Возврат нового множества (объекта класса Set).
}

// Удаление элемента из множества.
public static Set operator -(Set ob, char ch) {
   Set newset = new Set{);
```

```
    int i = ob.find(ch); // Если элемент не найден,
                         // неременной i будет присвоено значение -1.

    // Копирование оставшихся элементов с использованием перегруженного
    // оператора +.
    for(int j=0; j < ob.len; j++)
      if(j != i) newset = newset + ob.members[j];

    return newset;
  }

  // Объединение множеств.
  public static Set operator +(Set ob1. Set ob2) {
    Set newset = new Set(obi); // Копирование первого множества.

    // Добавление уникальных элементов из второго множества,
    for (int i=0; i < ob2.len; i++)
      newset = newset + ob2 ! 1] ;

    return newset; // Возврат нового множества.
  }

  // Разность множеств.
  public static Set operator -(Set ob1, Set ob2) {
    Set newset = new Set(ob1); // Копирование первого множества.

    // Вычитание из первого множества элементов второго множества,
    for(int i=0; i < ob2.len; i++)
      newset = newset — ob2[i];

    return newset; // Возврат нового множества.
  }

}

// Использование класса Set.
class SetDemo {
  public static void Main() {
    // Конструирование пустого множества.
    Set s1 = new Set();
    Set s2 = new Set();
    Set s3 = new Set();

    s1 = s1 + 'A';
    s1 = s1 + 'B';
    s1 = s1 + 'C';

    Console.Write("Множество s1 после добавления символов A, B и C: ");
    for(int i=0; i<sl.Length; i++)
      Console.Write(s1[i] + " ");
    Console.WriteLine();

    sl = sl - 'B';
    Console.Write("Множество sl после удаления символа B: ");
    for (int i=0; i<sl.Length; i++)
      Console.Write(sl[i] + " ");
```

```
Console.WriteLine();

sl = sl - 'A';
Console.Write("Множество sl после удаления символа A: ");
Lor (int i = 0; i<sl.Length; i++)
   Console.Write(sl[i] + " ");
Console.WriteLine();

sl = sl - 'C';
Console.Write("Множество sl после удаления символа C: ");
for(int i=0; i<sl.Length; i++)
   Console.Write(sl[i] + " ");
Console.WriteLine("\n");

s1 = s1 + 'A';
s1  = s1 + 'B';
s1  = s1 + 'C';
Console.Write("Множество s1 после добавления символов A, B и C: ");
for(int i=0; i<s1.Length; i++)
   Console.Write(s1[i] + " ");
Console.WriteLine();

s2  = s2 + 'A';
s2  = s2 + 'X';
s2  = s2 + 'W';

Console.Write("Множество s2 после добавления символов A, X и W: ");
for(int i=0; i<s2.Length; i++)
   Console.Write(s2[i] + " ");
Console.WriteLine();

s3 = s1 + s2;
Console.Write("Результат объединения множеств s1 и s2: ");
for(int i=0; i<s3.Length; i++)
   Console.Write(s3[i] + " ");
Console.WriteLine();

s3 = s3 — s1;
Console.Write("Множество s3 после выполнения операции s3 — s1: ");
for(int i=0; i<s3.Length; i++)
   Console.Write(s3[i] + " ");
Console.WriteLine("\n");

s2 = s2 + s2; // Удаление всех элементов множества s2.
s2 = s2 + 'C'; // Добавление символов A, B и C в обратном порядке.
s2 = s2 + 'B';
s2 = s2 + 'A';

Console.Write("Множество s1 теперь содержит символы: ");
for(int i=0; i<s1.Length; i++)
   Console.Write(s1[i] + " ");
Console.WriteLine();

Console.Write("Множество s2 теперь содержит символы: ");
for(int i=0; i<s2.Length; i++)
   Console.Write(s2[i] " ");
```

```
      Console.WriteLine();

      Console.Write("Множество s3 теперь содержит символы: ");
      for(int i=0; i<s3.Length; i++)
        Console.Write(s3[i] + " ");
      Console.WriteLine();
  }
}
```

Результат выполнения программы:

```
Множество s1 после добавления символов A, B и C: A B C
Множество s1 после удаления символа B: A C
Множество s1 после удаления символа A: C
Множество s1 после удаления символа C:

Множество s1 после добавления символов A, B и C: A B C
Множество s2 после добавления символов A, X и W: A X W
Результат объединения множеств s1 и s2: A B C X W
Множество s3 после выполнения операции s3 — s1: X W
Множество s1 теперь содержит символы: A B C
Множество s2 теперь содержит символы: C B A
Множество s3 теперь содержит символы: X W
```

Контрольные вопросы

1. Предложите общую форму синтаксиса, используемого при перегрузке операторов. Каков должен быть тип параметра метода `operator`?

2. Что нужно учесть при перегрузке оператора, чтобы его операндами могли быть объект необходимого класса и встроенный тип?

3. Может ли быть перегружен оператор `?` ? Можно ли изменить приоритет оператора?

4. Объясните, что такое индексатор. Приведите общую форму его синтаксиса.

5. Какие функции выполняют методы доступа `get` и `set` в индексаторе?

6. Что такое свойство? Каков его синтаксис?

7. Можно ли объявлять переменную в свойстве?

8. Может ли свойство передаваться в качестве аргумента `ref` или `out`?

9. Для созданного в проекте 7-1 класса `Set` определите операторы < и > таким образом, чтобы они проверяли, является множество подмножеством или супермножеством для другого множества. Причем оператор < должен возвращать значение `true`, если множество, указанное слева от оператора, является подмножеством множества, находящегося справа от оператора, и возвращать значение `false`, если это условие не выполняется. Оператор > должен возвращать значение `true`, если множество, указанное слева от оператора, является супермножеством для множества, находящегося справа от оператора, и возвращать значение `false`, если это условие не выполняется.

10. Для класса `Set` определите оператор `&`, выполняющий операцию пересечения двух множеств.

11. Попытайтесь определить другие операторы для класса `Set`. Например, попробуйте определить оператор `|`, выполняющий операцию «исключающее ИЛИ» для двух множеств. (Множество, являющееся результатом выполнения операции «исключающее ИЛИ», состоит из элементов, не принадлежащих пересечению этих множеств.)

Наследование

- Основы наследования
- Использование модификатора protected
- Вызов конструкторов наследуемого класса
- Использование ключевого слова base
- Создание многоуровневой иерархии классов
- Ссылки на объекты наследуемого и наследующего классов
- Создание виртуальных методов
- Использование абстрактных классов
- Применение ключевого слова sealed
- Класс object

Напомним, что C# является объектно-ориентированным языком и поддерживает основные принципы ООП, одним из которых является наследование. Используя наследование, вы можете создавать новые классы, представляющие собой расширение уже существующих классов. То есть класс может наследоваться другими классами, обладающими некими уникальными свойствами, которые добавляются к уже имеющимся характеристикам и возможностям. Например, можно создать класс *автомобиль,* имеющий такие общие характеристики, *как мощность двигателя,.расход топлива на 100 километров пробега, максимальная скорость, которую может развить автомобиль.* Этот класс могут наследовать два других класса — *грузовик,* в котором появится характеристика *грузоподъемность,* и *автобус,* в котором появится характеристика *количество перевозимых пассажиров.*

В языке C# класс, переменные и методы которого автоматически становятся членами нового создаваемого класса, называется *наследуемым* (*базовым) классом.* Класс, который к имеющимся членам наследуемого класса добавляет (определяет) новые члены класса, называется *наследующим* классом. Таким образом, наследующий класс является специализированной версией наследуемого класса. Наследующий класс наследует все переменные, методы, свойства и индексаторы, определенные наследуемым классом, добавляя при этом свои собственные уникальные элементы.

Основы наследования

В C# при объявлении наследующего класса указывается имя наследуемого класса. Рассмотрим небольшую программу, в которой демонстрируются ключевые свойства наследования. В ней создается класс `TwoDShape`, хранящий значения ширины и высоты двухмерного объекта, и наследующий класс `Triangle`. Обратите внимание на способ объявления класса `Triangle`.

```
// В программе создается простая иерархия классов.
using System;

// Класс, содержащий значения размеров двухмерной геометрической фигуры.
class TwoDShape {
  public double width;
  public double height;

  public void showDim() {
    Console.WriteLine("Значения ширины и высоты геометрической фигуры = " +
                      width + " и " + height);
  }
}

// Класс Triangle, наследующий класс TwoDShape.
class Triangle : TwoDShape {
  public string style;
  public double area() {
    return width * height / 2;
  }

  public void showStyie() {
    Console.WriteLine("Вид треугольника — " + style);
  }
```

Класс `Triangle` наследует класс `TwoDShape`. Обратите внимание на используемый синтаксис.

Класс `Triangle` может ссылаться на члены класса `TwoDShape` так же, как если бы они были объявлены непосредственно в классе `Triangle`.

```
}

   class Shapes {
  public static void Main() {
    Triangle t1 = new Triangle();
    Triangle t2 = new Triangle();

    t1.width = 4.0;
    t1.height = 4.0;
    t1.style = "равнобедренный.";

    t2.width = 8.0;
    t2.heignt = 12.0;
    t2.style = "прямоугольный.";

    Console.WriteLine("Информация об объекте t1: ");
    t1.showStyle();
    t1.showDim();
    Console.WriteLine("Площадь - " + t1.area());

    Console.WriteLine();

    Console.WriteLine("Информация об объекте t2: ");
    t2.showStyle();
    t2.showDim();
    Console.WriteLine("Площадь = " + t2.area());
```

Все члены класса `Triangle`, в том числе и те, которые были унаследованы из класса `TwoDShape`, доступны для объектов типа `Triangle`.

Результат выполнения программы:

```
Информация об объекте t1:
     Вид треугольника — равнобедренный.
Значения ширины и высоты геометрической фигуры — 4 и 4
Площадь = 8

  Информация об объекте t2:
Вид треугольника - прямоугольный.
Значения ширины и высоты геометрической фигуры - 8 и 12
Площадь = 48
```

В классе `TwoDShape` определены характеристики некоторой двухмерной геометрической фигуры, которой может быть квадрат, прямоугольник и т. д. В классе `Triangle`, который наследует класс `TwoDShape`, определяется специфический тип геометрической фигуры, в данном случае треугольник. В класс `Triangle` включаются все члены класса `TwoDShape` и добавляется поле `sryle`, а также методы `area()` и `showStyle()`. Название вида треугольника хранится в поле `style`, метод `area()` вычисляет и возвращает значение площади треугольника, а метод `showStyle()` отображает название вида треугольника.

Обратите внимание на то, что синтаксис, используемый при наследовании, достаточно прост. Имя наследуемого класса следует за именем наследующего класса, а между ними ставится двоеточие.

Поскольку в класс `Triangle` включены все члены наследуемого класса `TwoDShape`, его метод `area()` имеет доступ к значениям переменных `width` и `height`. Кроме

того, указав с использованием оператора точка (.) имя объекта t1 или t2 переменную width или height, можно непосредственно внутри метода Main обратиться к копиям этих переменных. На рис. 8.1 показано, каким образом TwoDShape включается в класс Triangle.

Несмотря на то, что класс TwoDShape является наследуемым для класса Triangle это полностью независимый автономный класс, который может использоваться самостоятельно. Например, действительным является следующий код:

```
TwoDShape shape = new TwoDShape{);

shape.width = 10;
shape.height = 20;

shape.showDim();
```

Объект класса TwoDShape не имеет доступа к любым членам наследующих классов. Синтаксис объявления класса, наследующего другой класс, приведен ниже.

```
class derived-class-name : base-class-name {
// тело класса
}
```

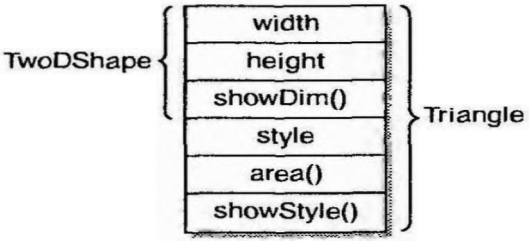


Рис. 8.1. Схема класса Triangle

Для каждого создаваемого наследующего класса можно указать лишь один наследуемый класс (наследование нескольких классов в C# не поддерживается, в отличии языка C++, поэтому при преобразовании кода C++ в код C# будьте очень внимательны). Однако в C# есть возможность создавать иерархию наследования, в котором наследующий класс сам может становиться наследуемым для другого класса наследовать сам себя класс не может.

Огромное преимущество наследования заключается в том, что после создания класса определяющего некоторые общие характеристики, он может использоваться создания любого количества наследующих классов со специфичными характеристиками. Каждый наследующий класс может разрабатывать свою собственную классификацию. Например, ниже приведен код еще одного класса, наследующего класса TwoDShape.

```
// В этом классе, который наследует класс TwoDShape, определена новая
// характеристика, присущая прямоугольникам.
Class Rectangle : TwoDShape {
  public bool isQguare() {
    if(width == height) return true;
    return false;
  }
```

```
  public double area()  {
    return width * height;
  }
}
```

В класс `Rectangle` включены все члены класса `TwoDShape`, а также метод `isSquare()`, определяющий, является ли прямоугольник квадратом, и метод `area()`, вычисляющий площадь прямоугольника.

Доступ к членам класса при использовании наследования

В главе 6 уже мы уже говорили, что члены класса часто определяются как закрытые для предотвращения их несанкционированного использования. При наследовании классов ограничения, имеющиеся при доступе к закрытым членам класса, не снимаются. Таким образом, метод наследующего класса не имеет доступа к членам наследуемого класса, если они являются закрытыми. Приведем код новой версии класса `TwoDShape`, в котором переменные `width` и `height` по умолчанию определяются как `private`, следовательно, метод `area()` класса `Triangle` не имеет к ним доступа.

```
// Эта программа не будет скомпилирована.
using System;

// Класс, содержащий значения размеров двухмерной геометрической фигуры.
Class TwoDShape {
  double width; // Теперь эти две переменные объявлены как закрытые,
  double height;

  public void showDim() {
    Console.WriteLine("Значения ширины и высоты геометрической фигуры = " +
                      width + " и " + height);
  }
}

// Класс Triangle наследует класс TwoDShape.
Class Triangle: TwoDShape {
  public string style;

  public double area ()  {
    return width * height / 2; // Ошибка, метод не имеет доступа к этим
                               // переменным.
  }

        public void showStyle() {
    Console.WriteLine("Вид треугольника - " + style);
  }
}
```

Метод не имеет доступа к закрытым членам наследуемого класса.

Эта программа не будет скомпилирована, поскольку при попытке обращения метода `area()`, определенного в классе `Triangle`, к закрытым переменным `width` и `height` возникнет ошибка доступа. Как только члены класса `weight` и `height` становятся закрытыми, доступ к ним могут осуществлять только другие члены их собственного класса, а доступ членов наследующего класса становится невозможным.

Закрытый член класса недоступен для любого кода, определенного вне этого класса в том числе и кода наследующих классов.

На первый взгляд данное правило кажется серьезным ограничением, но в C# существуют различные решения, позволяющие его обойти. Во-первых, есть возможность использования членов класса, имеющих тип доступа `protected` (объявленных с модификатором `protected`), они будут описаны в следующем разделе. Во-вторых, для получения доступа к закрытым данным можно воспользоваться свойствами объявленными как `public`. В предыдущих главах уже рассказывалось о том, что программисты на C# обычно обеспечивают доступ к закрытым членам классу используя методы или преобразуя закрытые члены класса в свойства. Ниже приведен код еще одной версии класса `TwoDShape`, в которой члены класса `width` и `height` преобразованы в свойства.

```
  // В программе демонстрируется использование свойств для доступа к закрытым
     членам класса.
using System;

// Класс, содержащий значения размеров двухмерной геометрической фигуры.
class TwoDShape {
   double pri_width; // Объявление закрытых переменных.
   double pri_height;

   // Свойства width и height.
   public double width {
      get { return pri_width; }
      set { pri_width = value; }
    }

   public double height {
      get { return pri_height; }
      set { pri_height = value; }
    }

   public void showDim() {
      Console.WriteLine("Значения ширины к высоты геометрической фигуры - " +
                        width + " и " + height);
    }
 }

// В этом классе, наследующем класс TwoDShape, определены характеристики,
// присущие треугольнику.
class Triangle : TwoDShape {
   public string style;

   public double area() {
      return width * height / 2;
    }

   public void showStyle() {
      Console.WriteLine("Вид треугольника - " style);
    }
}

class Shapes2 {
```

Определение свойств `width` и `height`.

Допускается использование свойств `width` и `height`, поскольку они были объявлены как открытые.

```
public static void Main() {
   Triangle t1 = new Triangle();
   Triangle t2 = new Triangle();

   t1.width = 4.0;
   t1.height = 4.0;
   t1.style = "равнобедренный.";

   t2.width = 8.0;
   t2.height = 12.0;
   t2.style = "прямоугольный.";
   Console.WriteLine("Информация об объекте t1: ");
      t1.showStyle();
    t1.showDim();
   Console.WriteLine("Площадь — " + t1.area());

   Console.WriteLine();

   Console.WriteLine("Информаиия об объекте t2: ");
   t2.showStyle();
   t2.showDim();
   Console.WriteLine("Площадь = " + t2.area());
  }
}
```

Ответы профессионала

Вопрос. В программировании на Java используются термины «суперкласс» и «подкласс». Употребляются ли эти термины в C#?

Ответ. Класс, который в Java называется «суперкласс», в C# получил название «наследуемый класс». Класс, именуемым в Java «подкласс», n C# называется «наследующий класс». Конечно, эти термины являются взаимозаменяемыми и могут использоваться в обоих языках, но в данной книге мы и далее будем применять стандартные термины C#. В языке C++ также используются термины «наследуемый класс», «наследующий класс».

Минутный практикум

1. Как и где указывается наследуемый класс при объявлении наследующего класса?
2. Включаются ли в наследующий класс члены наследуемого класса?
3. Может ли наследующий класс получать доступ к закрытым членам своего наследуемого класса?

---

1. Имя наследуемого класса указывается после имени наследующего класса и отделяется от него двоеточием.

2. Да.

3. Нет.

Использование модификатора protected

В предыдущем разделе уже рассказывалось о том, что члены наследующего класса имеют доступа к закрытым членам наследуемого класса. Исходя из этого, можно предположить, что когда требуется обеспечить доступ к некоторым членам наследуемого класса со стороны наследующего класса, эти члены класса обязательно должны быть объявлены с модификатором public. Но в таком случае данные члены классы становятся доступными для любого кода, определенного вне наследуемого класса что в некоторых случаях совершенно нежелательно. К счастью, подобные выводы несколько преждевременны. В C# существует возможность создания *защищен члена класса,* что делает его открытым во всей иерархии наследования данного класса, но оставляет закрытым для кода, определенного вне этой иерархии.

Защищенный член класса создается с помощью указания модификатора protect. Если член класса объявляется как protected, то это равносильно объявлению данного члена класса как закрытого, пока этот класс не станет наследуемым какого-нибудь класса. При наследовании класса A классом Б члены класса A, объявленные как protected, становятся открытыми для кода класса Б, но остаются закрыты для любого другого кода, определенного вне данной иерархии (то есть если класс наследует класс Б, то его код получает доступ к защищенным членам класса А).

Ниже приводится простая программа, в которой используется модификатор protected.

```
// В программе демонстрируется использование модификатора protected.
using System;

class B {
  protected int i, j; // Эти переменные является открытыми для членов
                      // класса D.

  public void set(int a, int b) {
    i = a;
    j = b;
  }

  public void show() {
    Console.WriteLine (i + " " + j);
  }
}

class D : B {
  int k;  //  Закрытая  переменная.

  //Метод класса D имеет доступ к переменным i и j, определенным в классе
  public void setk() {
    k = i * j;
  }

  public void showk() {
    Console.WriteLine(k);
  }
}
class ProtectedDemo {
```

Для полей i и j указан модификатор protected.

Метод, определенный в классе D, имеет доступ к переменным i и j, поскольку они являются защищенными, но не закрытыми.

```
  public static void Main() {
  D ob = new D();

   ob.set(2, 3); // Метод set () доступен из объекта класса D.
   ob.show() ; //     Метод show() доступен из объекта класса D.

   ob.seek(); // Эти обращения к методам экземпляра также действительны.
   ob.showk();
  }
}
```

Поскольку в этой программе класс `B` наследуется классом `D`, а переменные `i` и `j` объявлены в классе `B` как `protected`, метод `setk()` имеет к ним доступ.

Как и при использовании модификаторов `public` и `private`, модификатор `protected` остается присвоенным члену класса независимо от количества задействованных уровней наследования.

Конструкторы и наследование

В иерархии наследования допускается, чтобы наследуемые и наследующие классы одновременно имели свои собственные конструкторы. Но при этом возникает вопрос, какой конструктор (наследуемого класса, наследующего класса или оба) будет отвечать за создание объекта наследующего класса. Ответить на этот вопрос можно так — каждый из конструкторов создает (инициализирует) свою часть объекта. Это разделение функций имеет смысл, поскольку наследуемый класс не имеет доступа к какому-либо члену наследующего класса, в результате объект создается по частям. В предыдущих примерах автоматически вызывались конструкторы, заданные по умолчанию, но это не лучший способ создания объектов. Как правило, в большинстве классов су шествуют свои конструкторы.

Если в наследующем классе определен конструктор, процесс создания объекта не представляет особых сложностей, просто конструируется объект наследующего класса. Часть объекта, определенная в наследуемом классе, создается автоматически с помощью его конструктора, заданного по умолчанию. Ниже приводится модифицированная версия класса `Triangle`, в котором определен конструктор. При этом создается закрытое поле `style`, инициализируемое с помощью конструктора.

```
   // В этой программе в класс Triangle добавлен конструктор.
using System;

// Класс, содержащий значения размеров двухмерной геометрической фигуры.
 class TwoDShape {
 double pri_width; // Объявление закрытых переменных.
  double pri_height;

     // Свойства width и height.
  public double width {
       get { return pri_width; }
 set { pri_width - value; }
 }
 public double height {
       get { return pri_height; }
```

```
    set { pri_height - value; }
  }

  public void showDim() {
    Console.WriteLine("Значения ширины и высоты геометрической фигуры: " +
                      width + " и " + height);
  }
}

// В этом классе, наследующем класс TwoDShape, определены характеристики,
// свойственные треугольнику.
class Triangle : TwoDShape {
  string style; // Закрытая переменная.

  // Конструктор.
  public Triangle(string s, double w, double h) {
    width = w; // Инициализация свойств, определенных в наследуемом класса
    height = h;

    style = s;    // Инициализация переменной, определенной в наследующем классе
  }

  public double area() {
    return width * height / 2;


  public void showStyle()    {
    Console.WriteLine("Вид треугольника — " + style);
  }
}

class Shapes3 {
  public static void Main() {
    Triangle t1 = new Triangle ("равнобедренный.", 4.0, 4.0);
    Triangle t2 = new Triangle("прямоугольный.", 8.0, 12.0);

    Console.WriteLine("Информация об объекте t1: ");
    t1.showStyle();
    t1.showDim();
    Console.WriteLine("Площадь = " — t1.area());

    Console.WriteLine();

    Console.WriteLine("Информация об объекте t2: ");
    t2.showStyle();
    t2.showDim();
    Console.WriteLine("Площадь = " + t2.area());
  }
}
```

Конструктор класса `Triangle` инициализирует объявленное в нем поле `style` также наследуемые члены класса `TwoDShape`.

Если конструкторы определены и в наследуемом, и в наследующем классах, то процесс создания объекта несколько усложняется, поскольку в таком случае вызываются конструктора. В подобной ситуации нужно применять ключевое слово `base`, которое

используется и для вызова конструктора наследуемого класса, и для обеспечения доступа к члену наследуемого класса, если он был скрыт при объявлении члена наследуемого класса.

Вызов конструкторов наследуемого класса

Используя расширенную форму объявления конструктора наследующего класса, в котором указывается ключевое слово `base`, можно вызывать конструктор, определенный в его наследуемом классе. Ниже представлен синтаксис этой расширенной формы объявления.

```
        derived-constructor(parameter-list) : base(arg-list) {
// Тело конструктора.

}
```

В перечне `arg-list` указываются любые аргументы, необходимые конструктору наследуемого класса. Обратите внимание на то, как используется двоеточие.

Для приобретения навыков применения ключевого слова `base` используем в следующей программе еще одну версию класса `TwoDShape`, в котором определен конструктор, инициализирующим свойства `width` и `height`.

```
// В программе используется новая версия класса TwoDShape, в который добавлен.
// конструктор.
using system;

// Класс, содержащий значения размеров двухмерной геометрической фигуры.
class TwoDShape {
 double pri_width; // Объявление закрытых переменных.
 double pri_height;

 // Конструктор класса TwoDShape.
 public TwoDShape (double w, double h) {
   width = w;
   height = h;
 }

    // Свойства width и height.
 public double width {
   get { return pri_width; }
        { pri_width * value; }
 }

 public double height {
       get { return pri_height; }
   set { pri_height = value; }
 }

 public void snowDim () {
   Console.WriteLlne("Значения ширины и высоты геометрической фигуры — " +
                        width + " и " + height);
  }
}

    // В этом классе, наследующем класс TwoDShape, определены характеристики,
  // присущие треугольнику.
```

```
class Triangle : TwoDShape {
  string style; // Закрытая переменная.

  // Вызов конструктора наследуемого класса.
  public Triangle(string s, double w, double h) : base(w, h) {
    style = s;
  }

  public double area() {
    return width * height / 2;
  }

  public void showStyle() {
    Console.WriteLine("Вид треугольника - " + style);
  }
}

class Shapes4 {
  public static void Main() {
    Triangle t1 = new Triangle("равнобедренный.", 4.0, 4.0);
    Triangle t2 = new Triangle("прямоугольный.", 8.0, 12.0);

    Console.WriteLine("Информация об объекте t1: ");
    t1.showStyle();
    t1.showDim();
    Console.WriteLine("Площадь = " + t1.area());

    Console.WriteLine();

    Console.WriteLine("Информация об объекте t2: ");
    t2.showStyle();
    t2.showDim();
    Console.WriteLine("Площадь = " + t2.area{());
  }
}
```

Применение ключевого слова `base` для вызова конструктора `TwoDShape`.

В этой программе конструктор класса `Triangle()` с помощью ключевого слова `base` вызывает конструктор класса `TwoDShape` с параметрами `w` и `h`. В результате выполнения конструктора `base(w, h)` происходит инициализация свойств `width` и `height` указанными аргументами. То есть теперь конструктору класса `Triangle` не нужно инициализировать эти свойства, требуется лишь выполнить инициализацию переменной `style`, которая определена только в наследующем классе.

Наследуемый класс может иметь несколько конструкторов, каждый из которых можно вызывать с помощью ключевого слова `base`. Вызывается тот конструктор, список параметров которого соответствует указанному списку аргументов. Ниже приводится код программы, в которой используются расширенные версии классов `TwoDShape` и `Triangle`. В этих классах определены конструкторы по умолчанию и конструкторы, принимающие один или несколько аргументов.

```
// В программе используются расширенные версии классов TwoDShape и Triangle.
using System;

class TwoDShape {
  double pri_width; // Объявление закрытых переменных.
```

```
  double pri_height;

  // Определение конструктора по умолчанию.
  public TwoDShape() {
    width = height = 0.0;
  }

   // Конструктор с двумя параметрами.
  public TwoDShape(double w, double h) {
    width = w;
    height = h;
  }

  // Конструктор, предназначенный для создания объектов, у которых свойствам
  // width и height присваивается одно и то же значение.
  public TwoDShape(double x) {
    width = height = x;
  }

  // Определение свойств width и height.
  public double width {
    get { return pri_width; }
    set { pri_width = value; }
  }

  public double height {
    get { return pri_height; }
    set { pri_height = value; }
  }

  public void showDim() {
    Console.WriteLine("Значения ширины и высоты геометрической фигуры = " +
                      width + " и " + height);
  }
}

// В этом классе, наследующем класс TwoDShape, определены характеристики,
// свойственные треугольникам.
class Triangle : TwoDShape {
  string style; // Закрытая переменная.

  /* Конструктор по умолчанию. Он автоматически вызывает конструктор по
     умолчанию класса TwoDShape.
   */
  public Triangle() {
    style = "нуль.";
  }

  // Конструктор с тремя параметрами.
  public Triangle(string s, double w, double h) : base(w, h) {
    style = s;
  }

  // Конструктор, предназначенный для создания объекта, в котором определены
  // характеристики, свойственные равнобедренным треугольникам.
  public Triangle(double x) : base(x) {
```

Использование ключевого слова `base` для вызова различных конструкторов класса `TwoDShape`.

```
     style — "равнобедренный.";
   }

   public double area()b{
      return width * height / 2;
   }

   public void showStyle() {
     Console.WriteLine("Вид треугольника - " + style);
   }
}

class Shapes5 {
   public static void Main() {
      Triangle t1 = new Triangle();
      Trianqle t2 = new Triangle("прямоугольный.", 8.0, 12.0);
      Triangle t3 = new Triangle(4.0);

         t1 - t2;

      Console.WriteLine("Информация об объекте t1: ");
      t1.showStyle ();
      t1.showDim();
      Console.WriteLine("Площадь - " + t1.area());

Console.WriteLine();

      Console.WriteLine("Информация об объекте t2: ");
      t2.showStyle();
      t2.showDiE();
      Console.WriteLine( "Площадь — " + t2.area());

      Console.WriteLine():

      Console.WriteLine("Информация об объекте t3: ");
      t3.showStyle();
      t3.showDim();
      Console.WriteLine("Площадь = " + t3.area());

      Console.WriteLine();
   }
}
```

Результаты выполнения программы приведены ниже.

```
Информация об объекте t1:
Вид треугольника - прямоугольный.
Значения ширины и высоты геометрической фигуры = 8 и 12
Площадь = 48

Информация об объекте t2:
Вид треугольника прямоугольный.
Значения ширины и высоты геометрической фигуры = 8 и 12
Площадь = 48

Информация об объекте t3:
Вид треугольника — равнобедренный.
```

```
Значения ширины и высоты геометрической фигуры = 4 и 4
Площадь = 8
```

При указании в конструкторе наследующего класса ключевого слова `base` вызывается конструктор наследуемого класса. То есть ключевое слово `base` всегда ссылается на наследуемый класс, который в иерархии наследования находится непосредственно над вызывающим классом (вызывается конструктор класса, который непосредственно был наследован вызывающим классом). Это справедливо даже для многоуровневой иерархии. Передача аргументов наследуемому конструктору осуществляется путем указания этих аргументов в скобках после ключевого слова `base`. Если в конструкторе вызывающего класса отсутствует ключевое слово `base`, то автоматически вызывается конструктор наследуемого класса, определенный по умолчанию.

Минутный практикум

1. Каким образом из конструктора наследующего класса вызывается конструктор наследуемого класса?
2. Могут ли параметры передаваться конструктору наследуемого класса?
3. Всегда ли ключевое слово `base` ссылается на конструктор непосредственно наследуемого класса?

Скрытие переменных и наследование

В C# члены наследующего и наследуемого классов могут иметь одно и то же имя. В таком случае член наследуемого класса скрыт для других членов наследующего класса, и компилятор выводит предупреждение об этом на экран. Если вы намеренно скрываете член наследуемого класса и не хотите, чтобы компилятор выводил предупреждающее сообщение, то укажите перед членом наследующего класса ключевое слово new. Имейте в виду, что предназначение (выполняемые функции) ключевого слова new в данном случае отличается от его предназначения при создании экземпляра объекта.

Ниже приводится программа, в которой демонстрируется скрытие переменной при наследовании.

```
// В программе демонстрируется скрытие переменной при наследовании.
using System;

class A {
 public int i = 0;


// Наследующий класс.
class B : A {
 new int i; // Эта переменная i скрывает переменную i, объявленную в классе A.

 public B(int b) {
```

Переменная `i` в классе B скрывает переменную `i`, объявленную в классе A. Обратите внимание на использование ключевого слова `new`.

1. Для вызова конструктора наследуемого класса в конструкторе наследующего класса указывается ключевое слово `base`.

2. Да.

3. Да.

```
    i = b; // Инициализация переменной i в классе B.
  }

  public void, show (){
    Console.WriteLine("Значение переменной i в наследующем классе: " + i);
  }
}

class NameHiding {
  public static void Main() {
    B ob = new B(2);

    ob.show();
  }
}
```

Обратите внимание на использование ключевого слова `new`. Фактически оно указывает компилятору, что программист при создании новой переменной `i` намеренно скрывает переменную `i`, определенную в наследуемом классе `A`. Если ключевое слово `new` не будет указано, то компилятор выведет предупреждающее сообщение.

Результат работы этой программы выглядит следующим образом:

```
Значение переменной i в наследующем классе: 2
```

После определения переменной `i` в классе в будет скрыта переменная `i`, определенная в классе `A`. То есть при вызове метода `show()` для работы с данными объекта, имеющего тип `B`, будет отображаться значение переменной `i`, определенной в классе `B`.

Использование ключевого слова `base` для доступа к скрытой переменной

Ранее уже говорилось, что ключевое слово `base` применяется также для доступа к скрытым членам наследуемого класса. Используется следующая форма синтаксиса:

```
base.member
```

Здесь слово `member` представляет метод или переменную экземпляра. При таком использовании ключевое слово `base` выполняет практически те же функции, что и ключевое слово `this`, за исключением того, что слово `base` всегда ссылается на наследуемый класс.

Использование ключевого слова `base` для доступа к скрытой переменной демонстрируется в следующей программе:

```
// В программе демонстрируется использование ключевого слова base для
// доступа к скрытой переменной.
using System;

class A {
  public int i = 0;
}

// Создание наследующего класса.
class B : A {
  new int i; // Эта переменная i скрывает переменную i, объявленную в классе A.
  public B(int a, int b) {
```

```
    base.i = a; // Обращение к переменной i, определенной в классе A,
                // с помощью ключевого слова base и оператора точка (.).
    i = b;      // Инициализация переменной i, определенной в классе B.
   }

  public void show() {
    // Отображение переменной i, определенной в классе A.
    Console.WriteLine("Значение переменной i в наследуемом классе: " + base.i);

    // Отображение переменной i, определенной в классе B.
    Console.WriteLine("Значение переменной i в наследующем классе: " + i);
  }
 }

class UncoverName {
  public static void Main() {
    B ob = new B(1, 2);

    ob.show();
  }
}
```

Результат выполнения программы:

```
Значение переменной i в наследуемом классе: 1
Значение переменной i в наследующем классе: 2
```

Хотя переменная экземпляра `i` в объекте `ob` скрывает переменную `i`, определенную в классе `A`, использование ключевого слова `base` обеспечивает доступ к переменной `i`, определенной в наследуемом классе.

С помощью ключевого слова `base` также можно вызывать скрытые методы. Например,

```
// В программе демонстрируется вызов скрытого метода с помощью ключевого слова
// base.
using System;

class A {
  public int i = 0;

  // Метод show(), определенный в классе A.
  public void show() {
    Console.WriteLine("Значение переменной i в наследуемом классе: " + i);
  }
}

// Создание наследующего класса.
class B : A {
  new int i; // Эта переменная i скрывает переменную i, объявленную в классе A.

  public B(int a, int b) {
    base.i = a; // Обращение к переменной i, определенной в классе A, с помощью
                // ключевого слова base,
    i = b;      // Инициализация переменной i, определенной в классе B.
  }
```

```
  // Этот метод show() скрывает метол show(),
  // определенный в классе A.
  new public void show() {
    base.show(); // Вызов метода show(), определенного в классе A.

    // Вывод значения переменной i, инициализированной в классе B.
    Console.WnteLine("Значение переменной i в наследующем классе: " + i);
  }
}

class UncoverName {
  public static void Main() {
    B ob = new B(1, 2);

    ob.show();
  }
}
```

Метод `show()` скрывает метод с таким же именем, определенный в классе A.

Этот оператор вызывает скрытый метод `show()`.

Результат выполнения программы:

```
Значение переменной i в наследуемом классе: 1
Значение переменной i в наследующем классе: 2
```

Оператор `base.show();` вызывает метод `show()`, определенный в наследуемом классе.

Ключевое слово `new` указывает компилятору, что при создании нового метода `show()` программист намеренно скрывает метод `show()`, определенный в наследуемом классе A.

Минутный практикум

1. В наследующем классе объявляется переменная, которая будет скрывать переменную с таким же именем, определенную в наследуемом классе. Какое ключевое слово указывается перед именем переменной и ее модификатором при ее объявлении?
2. Какое ключевое слово используется для обращения к скрытой переменной (скрытому методу) наследуемого класса?
3. Можно ли из наследующего класса вызывать скрытый метод, определенный в наследуемом классе?

Проект 8.1. Расширение возможностей класса `Vehicle`

`TrucKDemo.cs`

В этом проекте мы займемся усовершенствованием и наследованием класса `Vehicle`, разработанного в главе 4. Напомним, что переменные класса `Vehicle` содержат следующую информацию об автомобиле — количество пассажиров, которое может

1. `new`.
2. `base`.
3. Да.

перевезти автомобиль, запас топлива, а также расстояние (в милях), которое этот автомобиль может проехать, используя один галлон топлива. Воспользуемся классом Vehicle как базовым для разработки более специализированных классов, например для разработки класса Truck, который бы хранил информацию о таком типе автомобили, как грузовик. Важнейшей характеристикой грузовика является грузоподъемность, поэтому для создания класса Truck необходимо только наследовать класс Vehicle и добавить в определение класса конструктор, инициализирующий переменную, в которой будет храниться значение грузоподъемности. Чтобы усовершенствовать класс Vehicle, объявим его переменные закрытыми и определим свойства для доступа к этим переменным.

Пошаговая инструкция

1. Создайте файл TruckDemo.cs и скопируйте в него код последней версии класса Vechicle из главы 4.
2. Создайте класс Truck, как показано ниже.

```
// Определение класса Truck, наследующего класс Vehicle.
class Truck : Vehicle {
   int pri_cargocap; // Грузоподъемность (в фунтах).

   // Конструктор класса Truck.
   public Truck (int p, int f. int :n, int c) : base (p, t, m)
   {
      cargocap = c;
   }

   // Свойство, обеспечивающее доступ к переменной pri_cargocap.
   public int cargocap {
      get { return pri_cargocap; }
      set { pri_cargocap = value; }
   }
}
```

Класс Truck наследует класс Vehicle, причем добавляется свойство cargocap. Таким образом, в класс Truck включена вся информация о транспортном средстве, определенная с помощью класса Vehicle. Добавлять в класс Truck следует только те члены класса, которые придают ему специфические характеристики, делая его уникальным.

3. Объявите переменные класса Vehicle закрытыми, а затем переименуйте их.

```
int ppi_passengers; // Количество пассажиров.
int pri_fuelcap; // Емкость топливного бака (в галлонах).
int pri_mpg; // Расстояние (в милях), которое данный автомобиль
                        // может проехать, используя один галлон топлива.
```

4. Добавьте свойства, определяющие доступ к этим переменным.

```
// Свойства.
public int passengers {
   get { return pri_passengers; }
   set { pri_passengers = value; }
}

 public int fuelcap {
```

```
    get { return pri_fuelcap; }
    set { pri_fuelcap = value; }
  }

  public int mpg {
    get { return pri_mpg; }
    set { pri_mpg = value; }
  }
```

5. Ниже приведен полный код программы, в которой на основе класса `Vehicle` с помощью наследования создается новый класс `Truck`.

```
/*
  Проект 8.1.

  В программе демонстрируется использование наследования для создания
  нового класса.
 */

using System;

class Vehicle {
  int pri_passengers;     // Количество пассажиров.
  int pri_fuelcap;        //  Емкость топливного бака (в галлонах).
  int prl_mpg;            //  Расстояние (в милях), которое данный автомобиль
                          // может проехать, используя один галлон топлива.

  // Конструктор класса Vehicle,
  public Vehicle(int p, int f, int m) {
    passengers = p;
    fuelcap = f;
    mpg = m;
  }

  // Метод возвращает значение максимального расстояния, которое может
  // проехать автомобиль с полным топливным баком.
  public int range() {
    return mpg * fuelcap;
  }

  // Метод вычисляет количество топлива, необходимое для преодоления
  // расстояния, значение которого передается методу в качестве параметра.
  public double fuelneeded(int miles) {
    return (double) miles / mpg;
  }

  // Свойства,
  public int passengers {
    get { return pri_passengers; }
    set { pri_passengers = value; }
  }

  public int fuelcap {
    get { return pri_fuelcap; }
    set { pri_fuelcap = value; )
  }
```

```
  public int mpg {
    get { return pri_mpg; }
    set { pri_mpg = value; }
  }
}

// Определение класса Truck, наследующего класс Vehicle.
 class Truck : Vehicle {
  int pri_cargocap; // Грузоподъемность (в фунтах).

  // Конструктор класса Truck.
  public Truck(int p, int t, int m, int c) : base(p, f, m)
  {
    cargocap = c;
  }

  // Свойство, обеспечивающее доступ к переменной pri_cargocap.
  public int cargocap {
    get { return pri__cargocap; }
    set { pri_cargocap = value; }
  }
}

class TruckDemo {
  public static void Main() {

    // Создание некоторых объектов класса Truck.
    Truck semi = new Truck(2, 200, 7, 44000); // Автофургон.
    Truck pickup = new Truck(3, 28, 15, 2000); // Пикап,
    double gallons;
    int dist = 252;

    gallons = semi.fuelneeded(dist);

    Console.WriteLine("Грузоподъемность автофургона = " + semi.cargocap +
    " фунтов.");
    Console.WriteLine("Чтобы проехать " + dist + " мили, автофургону " +
                      "требуется " + gallons + " галлонов топлива.\n");

    gallons = pickup.fuelneeded(dist);

    Console.WriteLine("Грузоподъемность пикапа = " + pickup.cargocap +
    " фунтов.");
    Console.WriteLine("Чтобы проехать " + dist + " мили, пикапу " +
                      "требуется " + gallons + " галлона топлива. \n") ;
  }
}
```

6. Результат выполнения программы:

```
Грузоподъемность автофургона = 44000 фунтов.
Чтобы проехать 252 мили, автофургону требуется 36 галлонов топлива.

Грузоподъемность пикапа = 2000 фунтов.
Чтобы проехать 252 мили, пикапу требуется 16,8 галлона топлива.
```

7. Класс Vehicle могут наследовать многие другие типы классов. Например, следующем фрагменте кода приведены первые строки определения класса, который содержит информацию о клиренсе внедорожного автомобиля.

```
// Создание класса, содержащего характеристики внедорожного автомобиле
  class OffRoad : Vehicle {
    int groundClearance; // Величина клиренса (в дюймах).

  // ...
}
```

Изучив вышеизложенный материал, сделаем выводы. Итак, используя наследование вы можете создавать новый класс, в котором определены общие черты какого-либо объекта, эти черты могут быть унаследованы новым классом, в который добавляются его уникальные характеристики.

Создание многоуровневой иерархии классов

До настоящего момента мы использовали простую иерархию классов, состоящую из наследуемого и наследующего классов. В C# можно также создавать иерархию, содержащую любое количество уровней наследования. Ранее говорилось, что можно использовать наследующий класс как базовый для дальнейшего наследования. В качестве примера рассмотрим три класса — A, B и C. Пусть класс с наследует класс в который в свою очередь наследует класс A. В подобной ситуации каждый наследующий класс наследует все черты, характерные для его наследуемого класса. В данном случае класс с наследует все характеристики классов B и A.

В следующей программе демонстрируется использование многоуровневой иерархии. Здесь класс Triangle уже является базовым при создании класса ColorTriangle, который наследует все характеристики классов Triangle и TwoDShape. В классе ColorTriangle объявлена новая переменная color, содержащая информацию о цвете треугольника.

```
// Многоуровневая иерархии классов.
using System;

class TwoDShape {
  double pri_width; // Объявление закрытых переменных,
  double pri_height;

  // Определение конструктора по умолчанию
  public TwoDShape() {
    width = height = 0.0;
  }

  // Конструктор с двумя параметрами.
  public TwoDShape(double w, double h) {
    width = w;
    height = h;
  }

  // Конструктор, предназначенный дли создания объектов, у которых свойствам
  // width и height присваивается одно и тоже значение.
  public TwoDShape(double x) {
```

```
    width = height - x;
   }

    // Определение свойств width и height.
  public double width {
    get { return pri_width; }
    set { pri_wiath = value; }
   }

  public double height {
     get { return pri_height; }
    set { pri_height = value; }
   }

  public void showDim() {
    Console.WriteLine("Значения ширины и высоты 1'еомегрической фигуры - "
        - width + " и " + height);
   }
}

// В этом классе, наследующем класс TwoDShape, определены характеристики,
// свойственные треугольникам.
class Triangle : TwoDShape {
  string style; // Закрытая переменная.

  /* Конструктор по умолчанию. Он автоматически вызывает конструктор
      по умолчанию класса TwoDShape. */

  public Triangle() {
    style = "нуль";
  }

  // Конструктор с тремя параметрами.
  public Triangle(string s, double w, double h) : base(w, h) {
    style = s;


  // Конструктор, предназначенный для создания объекта, в котором определены
     // характеристики, свойственные равнобедренным треугольникам.
  public Triangle(double x) : base(x) {
    style = "равнобедренный";
  }

  public double area() {
    return width * height / 2;
   }

  public void showStyle() {
    Console.WriteLine("Вид треугольника - " + style);
  }
}

   // Класс ColorTriangle,
   // наследующий класс Triangle.
class ColorTriangle : Triangle {
  string color;
```

Класс `ColorTriangle` наследует класс `Triangle`, который в свою очередь наследует класс `TwoDShape`, поэтому в классе `ColorTriangle` присутствуют все члены классов `Triangle` и `TvoDShape`.

```
  public ColorTriangle(string c, string s,
                double w, double h) : base(s, w, h) {
    color = c;
  }

  // Вывод информации о цвете треугольника.
  public void showColor() {
    Console.WriteLine("Цвет треугольника - " + color);
  }
}

class Shapes6 {
  public static void Main() {
    ColorTriangle t1 =
          new ColorTriangle("голубой", "прямоугольный", 8.0, 12.0);
    ColorTriangle t2 =
          new ColorTriangle("красный", "равнобедренный", 2.0, 2.0);

    Console.WriteLine("Информация об объекте t1: ");
    t1.showStyle();
    t1.showDim();
    t1.showColor();
    Console.WriteLine("Площадь = " + t1.area());

    Console.WriteLine();

    Console.WriteLine("Информация об объекте t2: ");
    t2.showStyle() ;
    t2.showDim();
    t2.showColor();
    Console . WriteLine ( "Площадь = " + t2.area());
  }
}
```

Объект класса `ColorTriangle` может вызывать методы, определенные в нем, а также в его наследуемых классах.

Результат выполнения программы:

```
Информация об объекте t1:
Вид треугольника — прямоугольный
Значения ширины и высоты геометрической фигуры = 8 и 12
Цвет треугольника — Голубой
Площадь = 48

Информация об объекте t2:
Вид треугольника — равнобедренный
Значения ширины и высоты геометрической фигуры = 2 и 2
Цвет треугольника - красный
Площадь = 2
```

Поскольку при создании класса `ColorTriangle` применяется многоуровневое наследование, объекты этого класса могут использовать все члены классов `Triangle` и `TwoDShape`. При определении класса `ColorTriangle` добавляется только строковая переменная, необходимая для его специфической цели — возможности указать цвет треугольника. Наследование обладает тем преимуществом, что позволяет повторно использовать код.

Отметим, что данная программа демонстрирует еще одну важную особенность - ключевое слово `base` всегда ссылается на конструктор ближайшего наследуемого

класса. Если ключевое слово base указывается в классе ColorTriangle, то вызывался конструктор класса Triangle. Если же ключевое слово base указывается в классе Triangle, то вызывается конструктор класса TwoDShape. Когда конструктору наследуемого класса необходимы какие-либо параметры, то в соответствии с правилами создания многоуровневой иерархии классов все наследующие классы должны предавать эти параметры «вверх» на соответствующий уровень, причем вне зависимости от того, нужны ли эти параметры самому наследующему классу.

Когда вызываются конструкторы

При изучении предыдущей темы мог возникнуть вполне закономерный вопрос о том, какой конструктор при создании наследующего класса вызывается первым — конструктор наследующего класса или конструктор наследуемого класса. В иерархии классов конструкторы вызываются в том порядке, в котором выполнялось наследование — от наследуемого класса к наследующему. Более того, этот порядок остается неизменным независимо от того, используется ключевое слово base или нет. Если это ключевое слово не указывается, вызывается заданный по умолчанию конструктор (без параметров) каждого наследуемого класса. Порядок вызова конструкторов демонстрируется в следующей программе:

```
// В программе демонстрируется порядок вызова конструкторов.
using System;

// Создание наследуемого класса.
class A {
  public A() {
    Console.WriteLine("Метод, определенный в классе A.");
  }
}

// Создание класса, наследующего класс A.
class B : A {
  public B() {
    Console.WriteLine("Метод, определенный в классе B.");
  }
}

// Создание класса, наследующего класс B.
      class C : B {
  public C() {
    Console.WriteLine("Метод, определенный в классе C.");
  }
}

class OrderOfConstruction {
  public static void Main() {

    C c = new C();

  }
}
```

В результате выполнения этой программы будут выведены следующие строки:

```
Метод, определенный в классе A.
Метод, определенный в классе B.
Метод, определенный в классе C.
```

Как видите, конструкторы вызываются в порядке наследования, и это оправдано, поскольку любая инициализация, выполняемая в наследуемом классе, может служить основой для инициализации, выполняемой наследующим классом. Поэтому конструктор наследуемого класса должен вызываться первым.

Ссылки на объекты наследуемого и наследующего классов

Как вы уже знаете, C# является языком, требующим строгого соблюдения типа (например, при выполнении операции присваивания). Автоматическое преобразование типа, выполняющееся при работе с переменны ми обычных типов, не распространяется на переменные ссылочного типа. То сеть ссылочная переменная одного, класса не может ссылаться на объект другого класса. Рассмотрим следующую программу:

```
// Эта программа не может быть скомпилирована.
class X {
  int a;

  public X(int i) { a = i; }
}

 class Y {
  int a;

  public Y(int i) { a = i; }
}

class IncompatibleRef {
  public static void Main() {
    X x = new X (10);
    X x2;
    Y y = new Y(5);

    x2 = x; // Эта операции присваивания действительна, поскольку в ней
            // участвуют ссылочные переменные одного типа.
    x2 = y; // Ошибка, типы ссылочных переменных не совпадают.
  }
}
```

Эти ссылочные переменные несовместимы.

В этой программе, даже если классы `x` и `y` физически совпадают, ссылочной переменной типа `X` невозможно присвоить ссылку на объект типа `Y`. То есть ссылочная переменная может ссылаться только на объекты одного типа.

Однако существует важное исключение в строгой типизации C#. Ссылочной переменной наследуемого класса может быть присвоена ссылка на объект любого наследующего класса. Рассмотрим, как выполняются эти операции.

```
// Ссылочная переменная наследуемого класса может ссылаться на объект
// наследующего класса.
using System;

class X {
  public int a;

  public X(int i) {
    a = i
  }
}

class Y : X {
  public int b;

  public Y(int i, int j) : base(j) {
    b = i;
  }
}

class BaseRef {
  public static void Main() {
    X x = new X (10);
    X x2;
    Y y = new Y(5, 6);

    x2 = x; // Это действительная операция, поскольку задействованы
            // переменные одного и того же типа.
    Console.WriteLine("Значение переменной x2.a: " + x2.a);

    x2 = y; // Также действительная операция,
            // поскольку класс Y наследует класс X.
    Console.WriteLine("Значение переменной x2.a: " + x2.a);

    // Ссылочная переменная типа X "знает" только о членах класса X.
    x2.a = 19; // Действительное обращение.
    // x2.b = 27; // Ошибка, в классе X не определена переменная b.
  }
}
```

Поскольку класс Y наследует кл
X, ссылочная переменная x2 м
жет ссылаться на объект y.

К этой программе класс `Y` наследует класс `x`; таким образом, переменной `x2` может быть присвоена ссылка на объект `Y`.

Важно понимать, что возможность доступа к членам класса зависит именно от типа ссылочной переменной, а не от типа объекта, на который она ссылается. То есть если ссылочной переменной, имеющей тип наследуемого класса, присваивается ссылка на объект наследующего класса, то с помощью этой переменной можно будет обращаться только к тем членам класса, которые были определены в наследуемом классе. Именно поэтому ссылочная переменная `x2` не имеет доступа к переменной экземпляра `b` даже после того, как переменной `x2` была присвоена ссылка на объект класса `Y`. Такое ограничение имеет смысл, поскольку наследуемый класс ничего не "знает" о тех членах класса, которые были добавлены при определении наследующего класса. Поэтому, чтобы приведенная выше программа могла быть скомпилирована ее последний оператор был закомментирован.

Довольно часто в классе определяется конструктор, который принимает в качестве параметра объект собственного класса. В результате класс может конструировать копию объекта. Также при вызове конструкторов в иерархии классов (при передаче конструкторам объектов в качестве параметров) ссылочной переменной, имеющей тип наследуемого класса, присваивается ссылка на объект наследующего класса, Например, обратите внимание на следующие версии классов `TwoDShape` и `Triangle`. В обеих версиях есть конструкторы, использующие объект в качестве параметра.

```
// Передача ссылочной переменной наследуемого класса ссылки на объект
// наследующего класса,
using System;

class TwoDShape {
  double pri_width; //Объявление закрытых переменных.
  double pri_height;

  // Определение конструктора по умолчанию.
  public TwoDShape() {
    width = height = 0.0;
  }

  // Конструктор с двумя параметрами.
  public TwoDShape(double w, double h) {
    width = w;
    height = h;
  }

  // Конструктор, предназначенный для создания объектов, у которых свойствам
  // width и height присваивается одно и то же значение.
  public TwoDShape(double x) {
    width = height = x;
  }

  // Конструирование объекта на основе другого объекта,
  public TwoDShape(TwoDShape ob) {
    width = ob.width;
    height = ob.height;
  }

  //Определение свойств width и height.
  public double width {
    get { return pri_width; }
    set { pri_width = value; }
  }

  public double height t
    get { return pri_height; }
    set { pri_height = value; }
  }

  public void showDim() {
    Console.WriteLine ("Значения ширины и высоты геометрической фигуры = " +
                       width + " и " + height);
```

Конструирование объекта на основе объекта этого же класса.

```
// В этом классе, наследующем класс TwoDShape, определены характеристики,
// свойственные треугольникам.
class Triangle : TwoDShape {
  string style; // Закрытая переменная.

  // Определение конструктора по умолчанию.
  public Triangle() {
    style = "нуль";
  }

  // Конструктор с тремя параметрами, вызывающий конструктор наследуемого
  // класса.
  public Triangle(string s, double w, double h) : base(w, h) {
    style = s;
  }

  // Конструктор, предназначенный для создания объекта, в котором определены
  // характеристики, присущие равнобедренным треугольникам.
  public Triangle(double x) : base(x) {
    style = "равнобедренный";


  // Конструирование объекта на основе другого объекта.
  public Triangle(Triangle ob) : base(ob) {
    style = ob.style;
    }

  public double area() {
    return width * height / 2;
  }

  public void showStyle() {
    Console.WriteLine("Вид треугольника - " + style);
  }


class Shapes7 {
  public static void Main() {
    Triangle t1 = new Triangle("прямоугольный", 8.0, 12.0);

    // Копия объекта t1.
    Triangle t2 = new Triangle(t1);

    Console.WriteLine("Информация об объекте t1: ");
    t1.showStyle();
    t1.showDim();
    Console.WriteLine("Площадь = " + t1.area());

    Console.WriteLine();

     Console.WriteLine("Информация об объекте — t2: ");
     t2.showStyle();
     t2.showDim();
     Console.WriteLine("Площадь = " + t2.area());
```

Передача ссылки на объект класса `Triangle` конструктору класса `TwoDShape`.

В программе объект t2 конструируется на основе объекта t1, следовательно, объект t1 и t2 идентичны. Результат выполнения программы приведен ниже:

```
Информация об объекта t1:
Вид треугольника - прямоугольный
Значения ширины и высоты геометрической фигуры - 8 и 12
Площадь - 43

Информация об объекте - t2:
Вид треугольника - прямоугольный
Значения ширины и высоты геометрической фигуры - 8 и 12
Площадь - 48
```

Обратите особое внимание на конструктор класса Triangle:

```
// Конструирование объекта на основе другого объекта.
public Trianqle(Triangle ob) : base(ob) {
   style - ob.style;
}
```

Этот конструктор принимает объект типа Triangle и передает его с помощью ключевого слова base конструктору класса TwoDShape. Приведем код этого конструктора:

```
// Конструирование объекта на основе другого объекта,
public TwoDShape(TwoDShape ob) X
   width = ob.width;
   height = ob.height;
}
```

Обратите внимание, что конструктор TwoDShape() ожидает получить в качестве, аргумента объект класса TwoDShape, но конструктор Triangle() передает ему объект класса Triangle. Как говорилось ранее, это происходит в силу того, что ссылочной переменной, имеющей тип наследуемого класса, может быть присвоена ссылка на объект наследующего класса. То есть передача ссылочной переменной ob, имеющей в конструкторе TwoDShape() тип TwoDShape, ссылки на объект ob, класса Triangle является действительной операцией. Поскольку конструктор TwoDShape() инициализирует только члены класса TwoDShape, его не «интересует» то, что объект может содержать другие члены класса, добавленные при определении наследующих классов.

Минутный практикум

1. Может ли наследующий класс использоваться в качестве базового класса для другого наследующего класса?
2. В каком порядке вызываются конструкторы в иерархии классов?
3. Предположим, что класс Jet наследует класс Airplane. Можно ли присвоить ссылочной переменной типа Airplane ссылку на объект типа Jet?

---

1. Да.

2. Конструкторы вызываются и порядке наследования.

3. Да. Во всех случаях ссылочная переменная тина наследуемого класса может ссылаться на объект наследующего класса, но не наоборот.

Виртуальные методы и переопределение

Метод, при определении которого в наследуемом классе было указано ключевое слово `virtual`, и который был переопределен (о *переопределении метода* будет рассказано в следующем абзаце) в одном или более наследующих классах, называется *виртуальным методом.* Следовательно, каждый наследующий класс может иметь собственную версию виртуального метода. В C# выбор версии виртуального метода, которую требуется вызнать, осуществляется в соответствии с типом объекта, на который ссылается ссылочная переменная. Этот выбор (определение) осуществляется *во время выполнения программы.* Ссылочная переменная может ссылаться па различные типы объектов, следовательно, могут быть вызваны различные версии виртуальных методов- другими словами, именно тип объекта, на который указывает ссылка (а не тип ссылочной переменной), определяет вызываемую версию виртуального метода. Таким образом, сели класс содержит виртуальный метод и от этого класса были наследованы другие классы, в которых определены свои версии метода, при ссылке переменной типа наследуемого класса на различные типы объектов вызываются различные версии виртуального метода. При определении виртуального метода в составе наследуемого класса перед типом возвращаемого значения указывается ключевое слово `virtual`, а при переопределении виртуального метода в наследующем классе используется модификатор `override`. Процесс определения виртуального метода внутри наследуемого класса, при котором частично или полностью изменяется тело метода, а имя, параметры и их типы остаются прежними, называется *переопределением метода.* Виртуальный метод не может быть определен с модификатором `static` или `abstract` (использование этих модификаторов рассматривается далее в этой главе).

Переопределение метода положено в основу концепции *динамического выбора вызываемого метода.* Это механизм, с помощью которого выбор вызываемого переопределенного метода осуществляется во время выполнения программы, а не во время компиляции.

Ниже приводится программа, в которой демонстрируется использование виртуального метода и его переопределенных версий.

```
// В программе демонстрируемся использование виртуального метода.
using System;

class Base {
  // Создание виртуального метода в наследуемом классе.
  public virtual void who() {
    Console.WriteLine("Метод who() , определенный в классе Base.");
  }
}

  class Derived1 : Base {
   // Переопределение метода who() в наследующем классе Derived1.
  dublic override void who() {
    Console.WriteLine("Метод who() , переопределенный в классе Derived1.");
   }
 }

class Derived2 : Base {
  // Еще одно переопределение метода who() в наследующем классе Derived2.
```

Объявление виртуального метода.

Переопределение виртуального метода.

```
  public override void who() {
    Console.WriteLine("Метод who(), переопределенный и классе Derived2.
  }
}

class OverrideDemo {
  public static void Main() {
    Base baseOb = new Base();
    Derived1 dOb1 = new Derived1();
    Derivea2 dOb2 = new Derived2();

    Base baseRef; // Объявление ссылочной переменной типа наследуемого класса

    baseRef = baseOb;
    baseRef.who();

    baseRef = dObl;
    baseRef.who();

    baseRef = dOb2;
    baseRef.who();
  }
}
```

Переопределение виртуального метода

В каждом случае вызываемая версия метода `who()` определяется во время выполнения программы и зависит от типа объекта» на который ссылается переменная `baseRef`.

Результат выполнения программы:

```
Метод who(), определенный в классе Base.
Метод who(), переопределенный в классе Derived1.
Метод who(), переопределенный в классе Derived2.
```

В этой программе создается наследуемый класс `Base`, а также два наследующих класса `Derived1` и `Derived2`. В классе `Base` объявляется метод `who()`, который затем переопределяется в его наследующих классах. В методе `Main()` объявляются объектом имеющие тип `Base`, `Derived1` и `Derived2`, а также объявляется ссылочная переменная `baseRef` типа `Base`. Затем ссылочной переменной `baseRef` поочередно присваиваются ссылки на объекты всех трех типов. Эти ссылки потом используются при вызове метода `who()`. Как видно из результата выполнения программы, выбор версии вызываемого метода `who()` зависит от типа объекта, на который ссылается переменная `baseRef`, а не от типа самой переменной.

Переопределять виртуальный метод не обязательно. Если наследующий класс не предоставляет собственную версию виртуального метода, то используется метод наследуемого класса. Например,

```
/* Если не переопределен виртуальный метод, используется метод наследуемого
   класса.
 */
using System;

class Base {
  // Создание виртуального метода в наследуемом классе.
  public virtual void who() {
    Console.WriteLine{"Метод who(), определенный в классе Base.");
  }
}

class Derived1 : Base {
```

```
  // Переопределение метода who() в наследующем классе.
  public override void who() {
    Console.WriteLine("Метод who(), переопределенный в классе Derived1.");
  }
}

class Derived2 : Base {
  // В этом классе не переопределяется метод who().
}

class NoOverrideDemo {
  public static void Main() {
    Base baseOb = new Base();
    Derived1 dOb1 = new Derived1();
    Derived2 dOb2 = new Derived2();

    Base baseRef; // Ссылочная переменная типа наследуемого класса.

    baseRef = baseOb;
    baseRef.who();
    baseRef = dOb1;
    baseRef.who();

    baseRef = dOb2;
    baseRef.who(); // Вызов метода who(), который был определен в классе Base.
  }
}
```

В этом классе не переопределяется метод `who()`.

Вызов метода `who()` класса `Base`.

Результат выполнения программы:

```
Метод who{), определенный в классе Base.
Метод who(), переопределенный в классе Derived1.
Метод who(), определенный в классе Base.
```

Здесь в классе `Derived2` не переопределяется метод `who()`, а при вызове метода `who()` объекта типа `Derived2` вызывается метод `who()`, который был определен в классе `Base`.

Для чего нужны переопределенные методы

Переопределенные методы обеспечивают в C# поддержку полиморфизма во время выполнения программы. Полиморфизм очень важная составляющая объектно-ориентированного программирования, позволяющая определять в наследуемом классе методы, которые будут общими для всех наследующих классов, при этом наследующий класс может определять специфическую реализацию некоторых или всех этих методов. Переопределение методов представляет еще один способ реализации в C# принципа полиморфизма «один интерфейс, несколько методов».

Чтобы успешно использовать полиморфизм, вы должны понимать, что наследующий и наследуемый классы формируют иерархию, в которой осуществляется переход от меньшей специализации к большей. В наследуемом классе определены члены класса, вторые могут непосредственно использоваться наследующим классом, и методы, которые в наследующем классе могут быть либо переопределены, либо оставлены без изменения. В результате наследующий класс получает определенную свободу при определении собственных версий методов, сохраняя при этом совместимый интерфейс

(то есть возможность обращения к версии метода с использованием одного и то же имени и одной и той же ссылочной переменной). Таким образом, применяя наследование и переопределение методов в наследуемом классе, вы можете определять общие формы (операторы и переменные) методов, которые могут использоваться всеми его наследующими классами.

Ответы профессионала

Вопрос. Могут ли свойства быть виртуальными?

Ответ. Да. Свойства могут быть объявлены с ключевым еловом `virtual`, затем переопределены с указанием ключевого слова `override`. Это справедливо для индекса горой.

Применение виртуальных методов

Продолжая изучать виртуальные методы, используем их в классе `TwoDShape`. В предыдущих программах для каждого класса, наследующего класс `TwoDShape`, определялся метод `area()`. Зная материал предыдущего раздела, мы можем усовершенствовать программу, определив в классе `TwoDShape` виртуальный метод `area()`. Этот метод можно переопределять в каждом наследующем классе, что дает возможность вычислять площадь в зависимости от типа геометрической фигуры (треугольник, квадрат и т. д.), информация о которой инкапсулирована в данном классе. (Для наглядности можно добавить в класс `TwoDShape` переменную `name` типа `string`, это позволит отображать название каждой геометрической фигуры.) Ниже приведен код этой программы.

```
// Использование виртуальных методов и полиморфизма.
using System;

class TwoDShape {
  double pri_width; // Объявление закрытых переменных,
  double pri_hcight;
  string pri_name;

  // Определение конструктора по умолчанию.
  public TwoDShape() {
    width = height - 0.0;
    name = "нуль";
  }

  // Конструктор с параметрами.
  public TwoDShape(double w, double h, string n) {
    width = w;
    height = h;
    name = n;
  }

  // Конструктор, предназначенный для создания объектов, у которых свойствам
  // width и height присваивается одно и то же значение.
  public TwoDShape(double x, string n) {
    width = height — x;
    name = n;
```

```
  }
    // Конструирование объекта на основе другого объекта.
  public TwoDShape(TwoDShape ob) {
    width = ob.width;
    height = ob.height;
    name = ob.name;
  }

  // Определение свойств width, height и name.
  public double width {
    jet { return pri_width; }
    set { pri_width = value; }
  }

  public double height {
    get { return pri_height; }
    set { pri_height = value; }
  }

  public string name {
    get { return pri_name; }
    set { pri_name = value; }
  }

  public void showDim() {
    Console.WriteLine("Значения ширины и высоты геометрической фигуры = " +
                      width + " и " + height);
  }

  public virtual double area() {
    Console.WriteLine("Этот метод переопределяется в наследующих классах.");
    return 0.0;
  }
}

// В этом классе, наследующем класс TwoDShape, определены характеристики,
// присущие треугольникам.
class Triangle : TwoDShape {
  string style; // Закрытая переменная.

   // Определение конструктора по умолчанию.
  public Triangle() {
    style = "нуль";
  }

   // Конструктор класса Triangle.
  public Triangle{string s, double w, double h) :
    base(w, h, "треугольник") {
      style = s;
  }

  // Конструктор, предназначенный для создания объекта, в котором определены
  // характеристики, присущие равнобедренным треугольникам.
  public Triangle(double x) : base(x, "треугольник") {
    style = "равнобедренный ";
```

Метод `area()` определен в классе `TwoDShape` как виртуальный.

```
  }

  // Конструирование объекта на основе объекта.
  public Triangle(Triangle ob) : base(ob) {
    style ob.style;


  // Переопределение метода area() в классе Triangle.
  public override double area() {
    return width * height / 2;
  }

  public void showStyle() {
    Console.WriteLine("Вид треугольника " + style);
  }
}

// Класс Rectangle, наследующий класс TwoDShape.
class Rectangle : TwoDShape {
  // Конструктор, предназначенный для создания объекта, в котором определены
  // характеристики, присущие прямоугольникам.
  public Rectangle(double w, double h) :
    base(w, h, "прямоугольник") { }

  // Конструктор, предназначенный для создания объекта, в котором определены
  // характеристики, присущие квадратам.
  public Rectangle(double x) :
    base(x, "квадрат") {   }

  // Конструирование объекта на основе другого объекта.
  public Rectangle(Rectangle ob) : base(ob) { }

  public bool isSquare() {
    if (width == height) return true;
    return false;
  }

  // Переопределение метода area() в классе Rectangle.
  public override double area() {
    return width * height;
  }
}

class DynShapes {
  public static void Main() {
    TwoDShape[] shapes = new TwoDShape[5];

    shapes[0] = new Triangle("прямоугольный", 8.0, 12.0);
    shapes[1] = new Rectangle(10);
    shapes[2] = new Rectangle(10, 4);
    shapes[3] = new Triangle(7.0);
    shapes[4] = new TwoDShape(10, 20, "базовая");

    for(int i=0; i < shapes.Length; i++) {
      Console.WriteLine("Эта геометрическая фигура называется — " +
        shapes[i].name);
```

Переопределение метода `area()` в классе `Triangle`.

Переопределение метода `area()` в классе `Rectangle`.

```
      Console.WriteLine("Площадь - " + shapes[i].area());

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

Результат выполнения программы:

```
    Эта геометрическая фигура называемся - треугольник
Площадь = 48

Эта геометрическая фигура называется - квадрат
Площадь = 100

Эта геометрическая фигура называется - прямоугольник
        Площадь = 40

Эта геометрическая фигура называется - треугольник
Площадь = 24,5

Эта геометрическая фигура называется - базовая
Это метод переопределяется в наследующих классах.
Площадь = 0
```

Теперь рассмотрим программу подробнее. Как говорилось ранее, сначала в классе `TwoDShape` объявляется метод `area()` с модификатором `virtual`, затем он переопределяется в классах `Triangle` и `Rectangle`. Метод `area()` в классе `TwoDShape` не содержит операторов, предназначенных для вычисления площади геометрической фигуры. В результате его выполнения просто выводится информация о том, что этот метод должен быть переопределен в наследующих классах, в которых он действительно должен будет производить вычисления. Поэтому при каждом переопределении метода `area()` в нем определяется оператор, подходящий для вычисления площади геометрической фигуры, характеристики которой инкапсулированы данным наследующим классом. Таким образом, в случае создания класса `Ellipse` (эллипс), метод `area()` должен будет вычислять площадь эллипса.

Обратите внимание, что в методе `Main()` объявлен массив `shapes`, содержащий объекты класса `TwoDShape`, но элементам данного массива присваиваются ссылки на объекты типа `Triangle`, `Rectangle` и `TwoDShape`. Эта правильно, поскольку ссылочные переменные типа наследуемого класса (которыми в данном случае являются элементы массива) могут ссылаться на объекты наследующего класса. Затем программа обрабатывает в цикле элементы массива, отображая сведения о каждом объекте. Хотя эта программа достаточно проста, она демонстрирует возможности наследования и переопределения методов. Тип объекта, ссылка на который хранится в ссылочной переменной, имеющей тип наследуемого класса, определяется во время выполнения программы, тогда же вызывается соответствующий метод. Если объект наследует класс `TwoDShape`, площадь геометрической фигуры вычисляется с помощью метода `area()`. Интерфейс этой операции одинаков для всех типов используемой фигуры (объекта).

Минутный практикум

1. Что такое виртуальный метод? Как он переопределяется?
2. Какой принцип C# поддерживают виртуальные методы?
3. Какая версия виртуального метода вызывается, если сам метод вызывается с помощью ссылочной переменной типа наследуемого класса?

Использование абстрактных классов

Иногда требуется создавать наследуемый класс, в котором определены лишь некоторые характеристики методов, такие как тип возвращаемого значения, имя и список параметров. Тела этих методов пустые, поскольку трудно предвидеть, какие переменные и операторы могут понадобиться в объектах наследующих классов. Поэтому в наследующем классе программист должен реализовать эти методы — определенные переменные и операторы, которые будут выполнять действия, необходимые для объекта этого класса. То есть в таком классе определяются лишь общие предназначения методов, которые должны быть реализованы в наследующих классах, но сам по себе этот класс не реализует один или несколько подобных методов. Например это происходит при использовании класса `TwoDShape`, рассмотренного в предыдущем разделе. Определение метода `area()` в классе `TwoDShape` представляет собой только «каркас», и при его выполнении ничего не вычисляется.

При создании библиотек классов вы, наверное, на собственном опыте убедились в том, что часто возникает ситуация, когда в наследуемом классе невозможно предвидеть действия, которые должен будет выполнять метод в наследующих класс В базовом классе метод можно определить лишь частично (например, в классе `TwoDShape` для метода `area()` выводится предупреждающее сообщение о том, что его нужно переопределить, чтобы он действительно возвращал значение площади фигуры). Но иногда, создав достаточно большую библиотеку классов или работа встроенными классами, программист вынужден (чтобы вспомнить нужную информацию либо сэкономить время) просматривать код класса или список членов классов. В этих списках приводятся лишь подписи методов — имена методов, типы возвращаемых значений, списки параметров. Если требуется наследовать этот класс, в такой информации недостаточно для понимания того, определен метод полностью или частично. Поэтому для более эффективного использования наследования базовом классе можно определить *абстрактные методы*, которые имеют пустые тела. При объявлении абстрактного метода используется модификатор `abstract`, поэтому встретив в списке членов класса метод с этим модификатором, программист знает что он *обязан* реализовать (определить) этот метод в наследующем классе. Абстрактный метод автоматически становится виртуальным, так что модификатор `virtual` при объявлении такого метода не нужен. Более того, совместное использовании модификаторов `virtual` и `abstract` приведет к возникновению ошибки.

---

1. Виртуальным называется метод, объявленный в наследуемом классе с помощью ключевой слова virtual и переопределенный в наследующем классе. Виртуальный метод переопределяется с указанием модификатора override.

2. Виртуальные методы являются одним из способов поддержки полиморфизма в C#.

3. Вызываемая версия переопределенного метода определяется исходя из типа объекта, тем который ссылается переменная во время вызова.

При объявлении абстрактного метода используется следующая форма синтаксиса:

```
abstract type name (parameter-list);
```

В этом объявлении отсутствует тело метода. Модификатор `abstract` может использоваться только с обычными методами и не может указываться для методов, имеющих тип `static`.

Класс содержащий один или более методов, должен быть объявлен как *абстрактный* (то есть при его объявлении указывается модификатор `abstract`). Поскольку абстрактный класс не определен полностью, объекты этого класса создать невозможно, и попытка создания объекта абстрактного класса с помощью ключевого слова `new` приведет к ошибке компиляции.

Когда класс наследует абстрактный класс, он должен реализовать все абстрактные методы наследуемого класса. Если этого не происходит, то наследующий класс также должен быть объявлен с модификатором `abstract`. Таким образом, атрибут `abstract` наследуется до тех пор, пока методы, а значит и сам класс, не будут полностью реализованы.

Продолжим модификацию класса `TwoDShape` и объявим его как абстрактный. В этом классе отсутствует реализация метода `area()`, предназначенного для вычисления площади неопределенной двухмерной геометрической фигуры. Поэтому в версии программы, которая рассматривалась в предыдущем разделе, метод `area()` класса `TwoDShape` определяется как абстрактный, а сам класс `TwoDShape` объявляется с модификатором `abstract`. Таким образом, наследование всеми классами класса `TwoDShape` означает, что в них должен быть переопределен метод `area()`.

```
// Создание абстрактного класса.
using System;

abstract class TwoDShape {
  double pri_width; // Объявление закрытых переменных.
  double pri_height;
  string pri_name;

  // Определение конструктора по умолчанию.
  public TwoDShape() {
    width = height = 0.0;
    name = "нуль";
  }

  // Конструктор с параметрами.
  public TwoDShape(double w, double h, string n) {
    width = w;
    height = h;
    name = n;
  }

  // Конструктор, предназначенный для создания объектов, у которых свойствам
  // width и height присваивается одно и то же значение.
        public TwoDShape (double x, string n) {
    width = height = x;
    name = n;
  }
```

Класс `TwoDShape` объявляется с модификатором `abstract`.

```
  // Конструирование объекта на основе объекта.
  public TwoDShape(TwoDShape ob) {
    width = ob.width;
    height = ob.height;
    name = ob.name;
  }

  // Определение свойств width, height и name.
  public double width {
    get { return pri_width; }
    set { pri_width = value; }
  }

  public double height {
    get { return pri_height; }
    set { pri_height = value; }
  }

  public string name {
    get { return pri_name; }
    set { pri_name = value; }
  }


  public void showDim() {
    Console.WriteLine("Значения ширины и высоты геометрической фигуры = " +
                      width + " и " + height);
  }

  // Теперь метод area() будет абстрактным.
  public abstract double area();
}

// В этом классе, наследующем класс TwoDShape, определены характеристики,
// присущие треугольникам.
class Triangle : TwoDShape {
  string style; // Закрытая переменная.

  // Определение конструктора по умолчанию.
  public Triangle() {
    style = "нуль";
  }

  // Конструктор класса Triangle.
  public Triangle(string s, double w, double h) :
    oase(w, h, "triangle") {
      style = s;
  }

  // Конструктор, предназначенный для создания объекта, в котором определены
  // характеристики, присущие равнобедренным треугольникам.
  public Triangle(double x) : base(x, "треугольник") {
    style = "равнобедренный";
  }

  // Конструирование объекта на основе объекта.
  public Triangle(Triangle ob) : base(ob) {
```

Объявление абстрактного метода `area()` в классе `TwoDShape`.

```
    style = ob.style;
  }

    // Переопределение метода area() в классе Triangle.
 public override double area() {
    return width * height / 2;
  }

         public void showStyle() {
    Console.WriteLine("Вид треугольника — " + style);
   }
}

// Класс Rectangle, наследующий класс TwoDShape.
class Rectangle : TwoDShape {
    // Конструктор, предназначенный для создания объекта, в котором определены
 // характеристики, присущие прямоугольникам.
 public Rectangle (double w, double h) :
   base (w, h, "прямоугольник") { }

 // Конструктор, предназначенный для создания объекта, в котором определены
 // характеристики, присущие квадратам.
 public Rectangle(double x) :
   class(x, "квадрат") { }

    // Конструирование объекта на основе объекта.
 public Rectangle(Rectangle ob) : base(ob) { }

 public bool isSquare() {
   if(width == height) return true;
   return false;
 }

    // Переопределение метода area() в классе Rectangle.
 public override double area () {
   return width * height;
  }
}

 class AbsShape {
 public static void Main() {
   TwoDShape[] shapes = new TwoDShape[4];

   shapes[0] = new Triangle("прямоугольный", 8.0, 12.0);
   shapes[1] = new Rectangle(10);
   shapes[2] = new Rectangle(10, 4);
   shapes[3] - new Triangle(7.0);

   for(int i=0; i < shapes.Length; i++) {
     Console.WriteLine("Эта фигура называется - " + shapes[i].name);
     Console.WriteLine("Площадь = " + shapes[i].area());

     Console.WriteLine();
   }
   }
```

Переопределение метода `area()` в классе `Triangle`.

Переопределение метода `area()` в классе `Rectangle()`.

Как видите, все наследующие классы должны либо переопределять метод `area()`, либо быть объявленными с модификатором `abstract`. Чтобы проверить это, попытайтесь создать наследующий класс, не переопределяющий метод `area()`. В результате будет выведено сообщение об ошибке компиляции. Конечно, остается возможность создания ссылочной переменной типа `TwoDShape`, что и реализовано в программе, но объявление объектов, имеющих тип `TwoDShape`, теперь невозможно. Поэтому в данной версии программы объявляется массив из четырех элементов, а не из пяти, как в предыдущей.

И последнее замечание, в класс `TwoDShape` по-прежнему включен метод `showDim()`, который не модифицирован с помощью ключевого слова `abstract`. В абстрактном классе могут присутствовать полностью определенные методы, которые могут использоваться наследующими классами без переопределения. В наследующих классах должны быть переопределены только абстрактные методы.

Минутный практикум

1. Что такое абстрактный метод? Как он создается?
2. Что такое абстрактный класс?
3. Можно ли создавать объекты абстрактного класса?

Использование ключевого слова `sealed` с целью предотвращения наследования

Каким бы полезным и мощным ни было наследование, иногда возникает необходимость в создании таких классов, которые наследовать невозможно. Например, если требуется класс, в котором инкапсулирована последовательность инициализации некоторых специализированных аппаратных средств, то нежелательно, чтобы пользователи класса имели возможность изменять способ инициализации монитора, поскольку в противном случае устройство может быть настроено неправильно.

В C# можно легко предотвратить возможность наследования класса с помощью ключевого слова `sealed`, которое указывается в начале объявления класса. Невозможно объявить класс, указав одновременно оба ключевых слова, `abstract` и `sealed`, поскольку абстрактный класс является незавершенным и для реализации своих абстрактных методов и возможности создания объектов *должен* быть наследован.

Пример класса, объявленного с использованием ключевого слова `sealed`:

```
sealed class A {
  //
}

// Это объявление класса недействительно.
class B : A {// ОШИБКА! Невозможно наследовать класс A
   //
}
```

Как следует из комментариев, класс `B` не может наследовать класс `A`, если класс `A` объявлен как `sealed`.

---

1. Абстрактным называется метод без тела, при создании которого указывается ключевое слово `abstract`, тип возвращаемого значения, имя метода и список параметров.

2. Абстрактным называется класс, который содержит как минимум один абстрактный метод.

3. Нет.

Класс object

В C# определен специальный класс object, который является базовым для всех других классов и для всех других типов (включая обычные типы). Другими словами, все принципы наследуют класс object, то есть ссылочная переменная типа object может ссылаться на объект любого другого типа. Также эта переменная может ссылаться на любой массив, поскольку массивы реализованы в C# как классы. Имя object — это краткая форма имени объекта System.Object, который является составной частью библиотеки классов .NET Framework.

В классе object определены следующие доступные для каждого объекта методы:

Метод	Назначение
`public virtual bool Equals(object object)`	Определяет, являются ли вызывающий объект и объект, передаваемый в качестве параметра *object*, одинаковыми
`public static bool Equals(object ob1, object ob2)`	Определяет, являются ли одинаковыми объекты *ob1* и *ob2*, переданные методу в качестве параметров
`protected Finalyze`	Выполняет действия, связанные с завершением выполнения программы, до «сборки мусора». В C# доступ к методу Finalize осуществляется с помощью деструктора
`public virtual int GetHashCode()`	Возвращает хеш-код, ассоциированный с вызывающим объектом
`public Type GetType()`	Получает тип объекта во время выполнения программы
`protected object MemberwiseClone()`	Создает «мелкую копию» объекта, в которую копируются члены класса, но при этом не копируются объекты, на которые ссылаются эти члены класса
`public static bool ReferenceEquals (object ob1, object ob2)`	Определяет, ссылаются ли ссылочные переменные *ob1* и *ob2* на один и тот же объект
`public virtual string ToString()`	Возвращает строку, описывающую объект

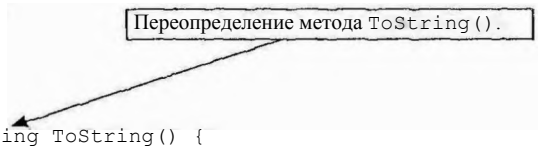
Описание назначения некоторых из указанных методов нуждается в дополнительных Пояснениях. Метод Equals(object) определяет, ссылается ли ссылочная переменная типа вызывающего объекта на тот же самый объект, что и ссылочная переменная, переданная в качестве аргумента (при этом определяется, будут ли совпадать две ссылки). При выполнении метода возвращается значение true, если это один и тот же объект, и значение false в противном случае. Этот метод может быть переопределен при создании пользовательских классов с указанием критериев, по которым будет оцениваться равенство объектов. Например, можно создать метод Equals(object), который сравнивает содержимое двух объектов. Метод Equals(object, object) для сравнения объектов, передаваемых в качестве параметров, вызывает метод Equals(object).

Метод GetHashCode() возвращает хеш-код, ассоциированный с вызывающим объектом. Этот код может использоваться произвольным алгоритмом, применяющим хеширование в качестве средства доступа к хранимым объектам.

В главе 7 уже говорилось, что для перегрузки оператора == обычно приходится переопределять методы `Equals()` и `GotHashCode()`, поскольку в большинстве случаев нужно, чтобы этот оператор и методы `Equals()` функционировали одинаково. В случае переопределения метода `Equals()` необходимо также переопределить метод `GetHashCode()` для достижения их совместимости.

Метод `ToString()` возвращает строку, содержащую описание объекта, из которого он вызывается. При указании объекта в качестве параметра метода `WriteLine()` метод `ToString()` вызывается автоматически. Во многих классах этот метод может быть переопределен, благодаря чему будет отображаться информация, относящаяся к каждому конкретному объекту данною класса. Например,

```
// В программе демонстрируется использование метода ToStrinq().
using System;

class MyClass {
  static int count = 1;
  int id;

  public MyClass() {
    id = count;
    count++;
  }

  public override string ToString() {
    return "Объект типа MyClass № " + id;
  }
}

class Test {
  public static void Main() {
    MyClass ob1 = new MyClass();
    MyClass ob2 = new MyClass();
    MyClass ob3 = new MyClass();

    Console.WriteLine(ob1);
    Console.WriteLine(ob2);
    Console.WriteLine(ob3);

  }
}
```



Результат выполнения программы:

```
Объект типа MyClass № 1
Объект типа MyClass № 2
Объект типа MyClass № 3
```

Упаковка и распаковка

Ранее уже говорилось о том, что все типы C#, включая обычные типы, наследуют класс `object`. Поэтому ссылочная переменная тина `object` может ссылаться на объект любого другого типа, включая обычные типы. При присваивании ссылочной переменной типа `object` значения обычного типа происходит процесс, называемый *упаковкой,* в результате выполнения которого значение будет храниться в объекте

типа object. То есть значение обычного типа будет «упаковано» внутри объекта. Затем этот объект может использоваться как любой другой объект. Упаковка выполняется автоматически, для этого нужно просто присвоить значение обычного типа ссылочной переменной типа object, остальную работу выполнит компилятор языка C#.

В процессе *распаковки* происходит извлечение значения из объекта. Для осуществления этого процесса требуется выполнение операции приведения типа. При присваивании обычной переменной значения, извлекаемого из объекта, перед ссылочной переменной типа object в скобках указывается тип, к которому должно быть прицелено распаковываемое значение.

Ниже приводится простая программа, в которой выполняется упаковка и распаковка значения.

```
// В программе демонстрируемся использование упаковки и распаковки.
 using System;

class BoxingDemo {
  public static void Main()
    itn = x;
    object obj;

    x = 10;
    obj = x; // Упаковка значения переменной x в объект.

    int y = (int)obj; // Распаковка (извлечение) значения, хранящегося в
                              // объекте obj, и присваивание его переменной типа int.

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

При выполнении этот оператора происходит упаковка значения переменной x в объект.

При выполнении этого оператора происходит распаковка значения.

При выполнении этой программы будет выведено значение 10. Обратите внимание, что упаковка значения переменной x выполняется в результате присваивания этого значения ссылочной переменной obj, имеющей тип object. Целочисленное значение извлекается из переменной obj при помощи выполнения операции приведения типа.

Ниже приводится еще один пример использования упаковки. В программе значение типа int передается в качестве аргумента методу sqr(), которому должен передаваться параметр типа object.

```
// В программе демонстрируется выполнение упаковки при передаче методу
// значений в качестве аргументов.
using System;

class BoxingDemo {
  public static void Main() {
    int x;
    x = 10;
    Console.WriteLine("Значение переменной x = " + x);

    // Значение переменной x автоматически упаковывается при передаче его
     // методу sqr().
    x = BoxingDemo.sqr(x);
    Console.WriteLine("Значение переменной x во второй степени = " + x);
  }
```

```
  static int sqr(object o) {
    return (int)o * (int)o;
  }
}
```

В результате выполнения программы будут выведены следующие строки:

```
Значение переменной x = 10
Значение переменной x во второй степени = 100
```

Значение переменной `x` автоматически упаковывается при передаче его методу `sqr()`. Упаковка и распаковка обеспечивают полную унификацию системы типов. Все типы наследуются из класса `object`. Ссылка на любой тип может быть присвоена ссылочной переменной. Упаковка/распаковка значений обычных типов выполняется автоматически. Кроме того, поскольку все типы наследуют класс object, они имеют доступ к методам класса `object`. В качестве примера приведем несколько неожиданную программу.

```
// При использовании упаковки можно вызывать методы значения так же, как
// обычно вызывают методы экземпляра!
using System;

class MethOnValue {
  public static void Main()      {

    Console.WriteLine(10.ToStnng());

  }
}
```

Такой оператор является действительным в C#!

В результате выполнения программы отображается строка, состоящая из двух символов 10. Это происходит в результате того, что метод `ToString()` возвращай строковое представление объекта, из которого он вызывается. В данном случае строковым представлением значения 10, принадлежащего к типу int, будет строка, состоящая из двух символов 10.

Минутный практикум

1. Если класс объявлен с модификатором `sealed`, может ли он быть наследуемым?
2. Что представляет собой класс `object`?
3. Как предотвратить возможность наследования класса?

1. Нет.

2. Класс `object` — это базовый класс для всех типов, определенных в C#.

3. Для предотвращения возможности наследования класс должен быть объявлен с модификатором `sealed`.

Контрольные вопросы

1. Имеет ли наследуемый класс доступ к членам наследующего класса? Имеет ли наследующий класс доступ к членам наследуемого класса?

2. Создайте наследующий класс класса `TwoDShape` и назовите его `Circle`. Включите в его определение метод `area()`, предназначенный для вычисления площади окружности, а также конструктор, использующий ключевое слово `base` для инициализации той части объекта, которая была наследована от класса `TwoDShape`.

3. Каким образом можно предотвратить доступ членов наследующего класса к членам наследуемого класса?

4. Укажите назначение ключевого слова `base`.

5. Для следующей иерархии наследования укажите порядок, в котором вызываются конструкторы классов при создании объекта класса `Gamma`.

```
class Alpha { ...
class Beta : Alpha { ...
class Gamma : Beta { ...
```

6. Ссылка наследуемого класса может указывать на объект наследующего класса. Объясните, почему это имеет значение при переопределении метода.

7. Что такое абстрактный класс?

8. Каким образом можно предотвратить возможность наследования класса?

9. Объясните, каким образом при использовании наследования, переопределения методов и абстрактных классов поддерживается полиморфизм.

10. Какой класс является базовым для всех остальных классов?

11. Объясните суть процесса упаковки.

12. Каким образом осуществляется доступ к членам класса, объявленным с модификатором `protected`?

Интерфейсы, структуры и перечисления

- Использование интерфейсов
- Применение ссылок на интерфейсы
- Добавление в интерфейсы свойств и индексаторов
- Наследование интерфейсов
- Использование явных реализаций
- Исследование структуры
- Применение перечислений

В настоящей главе рассматривается один из наиболее важных элементов C#: *интерфейс*. Этот элемент определяет набор методов, которые могут реализовываться с помощью класса. Конечно, интерфейс сам по себе не может реализовывать методы. Он является чисто логической конструкцией, которая описывает набор поддерживаемых классом методов, не диктуя при этом способ реализации.

В главе также обсуждаются два новых типа данных, поддерживаемых в C#: структуры и перечисления. Структуры напоминают классы, отличаясь от последних тем, что могу, обрабатывать типы значений, а не ссылочные типы. Перечисления представляют собой перечни именованных целочисленных констант. Благодаря этим новым типам данных значительно расширяются возможности среды программирования C#.

Интерфейсы

В объектно-ориентированном программировании иногда возникает необходимость определения действий, выполняемых классом, без указания способа выполнения этих действий. Ранее уже рассматривался подобный пример: абстрактный метод. Абстрактный тип метода определяет сигнатуру метода, но не поддерживает реализацию. В этом случае наследующий класс должен поддерживать свою собственную реализацию для каждого абстрактного метода, определенного его наследуемым классом. Таким образом, абстрактный метод определяет *интерфейс* метода, но не его *реализацию*. Исходя из полезности абстрактных классов и методов, имеет смысл расширить рассматриваемую концепцию. В C# можно полностью отделить интерфейс класса от реализации этого класса. В этом случае используется ключевое слово `interface`.

Синтаксис интерфейсов подобен синтаксису абстрактных классов. Однако в случае интерфейсов методы не могут включать тело. Таким образом, интерфейсы ни при каких условиях не поддерживают реализацию. Этот объект определяет, что нужно делать, но не демонстрирует, как именно это нужно делать. Сразу же после того, как интерфейс определен, он может реализовываться произвольным количеством классов. Один класс, в свою очередь, может реализовывать любое число интерфейсов.

Для выполнения реализации интерфейса класс должен поддерживать тела (реализации) для методов, описываемых с помощью данного интерфейса. Каждый класс может свободно определять детали своей реализации. В этом случае два класса могут Реализовывать один и тот же интерфейс различными способами, но каждый отдельный класс по-прежнему поддерживает один и тот же набор методов. Таким образом, код, который «знает» о наличии интерфейса, может использовать объекты любого класса в случае, если интерфейс для этих объектов будет одинаковым. Путем поддержки интерфейса C# позволяет полностью реализовать аспект полиморфизма, именуемый «один интерфейс, несколько методов».

Объявление интерфейсов выполняется с помощью ключевого слова `interface`. Ниже приводится упрощенная форма объявления интерфейса:

```
interface name {
  ref-type method-name1 (param-list);
  ref-type method-name2(param-list);
   // ...
ref-type method-nameN(param-list);

}
```

Имя интерфейса указывается с помощью параметра name. Объявление методом реализуется с помощью указания их сигнатур и типа возврата. Фактически в данном случае речь идет об абстрактных методах. Ранее уже упоминалось о том, что на интерфейсе, определенном с помощью ключевого слова `interface`, методы не могут иметь реализацию. Следовательно, каждый класс, включающий `interface`, должен реализовывать все эти методы. В интерфейсе для методов неявным образом задается тип `public`. В этом случае также не допускается явный спецификатор доступа.

Ниже приводится пример объекта `interface`. Он определяет интерфейс для класса который генерирует наборы чисел.

```
public interface Series {
  int getNext(); // Возврат следующего числа в наборе.
  void reset(); // Перезагрузка.
  void setStart(int x) ; // Установка начального значения.
}
```

При объявлении этого интерфейса используется модификатор `public`, поэтому он может реализовываться с помощью любого класса в любой программе.

В дополнение к объявлению сигнатур методов интерфейсы могут объявлять сигнатуры свойств, индексаторов и событий. События будут описываться в главе 10, и сейчас вкратце рассмотрим методы, свойства и индексаторы. Интерфейсы не могут включать данные-члены. Они также не могут определять конструкторы, деструкторы или операторные методы. Кроме того, никакой из членов не может определяться в виде `static`.

Реализация интерфейсов

Как только `interface` был определен, он может быть реализован одним или несколькими классами. Для реализации интерфейса после имени класса необходимой указать название интерфейса (почти так же, как и в случае с определением наследуемого класса). Ниже приведена общая форма синтаксиса кода класса, реализующего интерфейс:

```
class class-name: interface-name {
  // тело класса
}
```

Имя реализуемого интерфейса указывается с помощью параметра `interface-name`.

Если класс реализует интерфейс, он обязан реализовать его в целом. Например, невозможно указать и реализовать только часть интерфейса.

Классы могут реализовывать более одного интерфейса. В случае подобной реализации интерфейсы должны разделяться запятыми. Класс может наследовать наследуемый класс, а также реализовывать один или больше интерфейсов. В этом случае в списке, разделенном запятыми, имя наследуемого класса должно быть первым.

Методы, реализующие интерфейс, следует объявлять как `public`. Это необходимо делать по той причине, что методы неявным образом общедоступны внутри интерфейса, поэтому их реализации также должны быть общедоступны. При этом тип сигнатуры реализуемого метода должен точно соответствовать типу сигнатуры, указанному в определении `interface`.

Ниже приводится пример, в котором реализуется (с помощью класса `ByTwos` показанный ранее интерфейс `Series`. Этот класс позволяет генерировать набор чисел, причем каждое из последующих чисел на 2 больше, чем предыдущее.

```
// Реализация класса Series.
class ByTwos : Series {
 int start;
  int val;

  public ByTwos() {
    start = 0;
    val = 0;
  }

  public int getNext() {
    val += 2;
    return val;
   }

  public void reset() {
    start = 0;
    val = 0;
   }

  public void setStart(int x) {
    start = x;
    val = x;
   }
}
```

Реализация интерфейса `Series`.

Нетрудно заметить, что интерфейс `ByTwos` реализует все три метода, определенные с помощью интерфейса `Series`. Это пришлось сделать потому, что класс не может выполнить частичную реализацию интерфейса.

Ниже приводится код, демонстрирующий возможности класса `ByTwos`:

```
// Демонстрация возможностей класса ByTwos.
using System;

class SeriesDemo {
  public static void Main() {
    ByTwos ob = new ByTwos();

    for(int i=0; i < 5; i++)
      Console.WriteLine("Следующее значение " +
                        ob.getNext());

    Console.WriteLine("\nВосстановление");
    ob.reset();
    for(int i = 0; i < 5; i++)
      Console.WriteLine("Следующее значение " +
                        ob.getNext());

    Console.WriteLine{"\nНачало со 100");
    ob.setStart(100) ;
    for(int i=0; i < 5; i++)
      Console.WriteLine("Следующее значение " +
                        ob.getNext());
    }
  }
```

С целью компиляции класса SeriesDemo в процесс компиляции потребуется включить классы Series, ByTwos и SeriesDemo. В результате компилятор автоматически скомпилирует все три файла и создаст конечный исполняемый файл. Например, если данные файлы именуются Series.cs, ByTwos.cs и SenesDemo.cs, для запуска на выполнение процесса компиляции используется следующая командная строка:

```
>csc Series.cs ByTwos.cs SeriesDemo.cs
```

Если используется визуальная среда разработки Visual C++ IDE, тогда просто добавьте упомянутые три файла в проект C#. Помещение всех трех файлов в один и тот же файл вполне себя оправдывает.

Результат выполнения программы:

```
Следующее  значение  2
Следующее  значение  4
Следующее  значение  6
Следующее  значение  8
Следующее значение 10

Восстановление
Следующее  значение  2
Следующее  значение  4
Следующее  значение  6
Следующее  значение  8
Следующее значение 10

Начало  со  100
Следующее  значение  102
Следующее  значение  104
Следующее  значение  106
Следующее  значение  108
Следующее  значение  110
```

Классы, реализующие интерфейсы, могут определять дополнительные члены с целью их применения в своих собственных целях. Например, в следующей версии интерфейса ByTwos добавляется метод getPrevious(), возвращающий предыдущее значение:

```
// Реализация интерфейса Series и добавление метода getPrevious().
 class ByTwos : Series {
  int start;
  int val;
  int prev;

  public ByTwos()  {
    start = 0;
    val = 0;
    prev = -2;
   }

  public int getNext() {
    prev = val;
    val += 2;
    return val;
   }
```

```
  public void reset() {
    start = 0;
    val = 0;
    prev = -2;
  }

  public void setStart(int x) {
    start = x;
    val = x;
    prev = x - 2;
  }

  // Метод, не указанный с помощью Series.
    int qetPrevious(){
    return prev;
  }
}
```

Добавление метода, не определенною с помощью интерфейса Series.

Обратите внимание на то, что добавление метода `getPrevious()` потребует изменения реализаций методов, определенных с помощью интерфейса `Series`. Однако, поскольку интерфейс для этих методов остается неизменным, изменения будут не заметны и не отразятся па предшествующем коде. Это и является одним из преимуществ, обеспечиваемых интерфейсами.

Как объяснялось ранее, любое количество классов может быть реализовано посредством ключевого слова `interface`. Например, ниже приводится код класса `ByThrees`, который генерирует наборы чисел, кратных трем:

```
// Реализация интерфейса Series.
class ByThrees : Series {
  int start;
  int val;

  public ByThrees() {
    start = 0;
    val = 0;
  }

  public int getNext() {
    val += 3;
    return val;
  }

  public void reset() {
    start = 0;
    val = 0;
  }

  public void setStart(int x) {
    start = x;
    val = x;
  }
}
```

Реализация интерфейса Series другим способом.

Минутный практикум

1. Каково общее назначение интерфейса?
2. Какие элементы могут быть членами интерфейса?
3. Каким образом реализуются интерфейсы с помощью класса?

Использование интерфейсных ссылок

Вы, наверное, удивитесь, когда узнаете о том, что можно объявлять переменную ссылки, имеющую интерфейсный тип. Другими словами, можно создавать интерфейсную переменную ссылки. Подобная переменная может ссылаться на любой объект, который реализует ее интерфейс. При вызове метода объекта с помощью интерфейсной ссылки вызывается версия метода, реализуемого данным объектом Этот процесс напоминает использование ссылки наследуемого класса для доступа к объекту наследующего класса, как описано в главе 8.

В следующем примере иллюстрируется применение интерфейсной ссылки. Здесь задействована одна и та же интерфейсная переменная ссылки для вызова объектов в интерфейсах `ByTwos` и `ByThrees`.

```
// Демонстрация возможностей интерфейсных ссылок.
using System;

// Определение интерфейса
public interface Series {
   int getNext(); // Возврат следующего числа из набора.
   void reset(); // Перезапуск.
   void setStart(int x); // Установка начального значения.
}

// Реализация интерфейса Series одним способом.
class ByTwos : Series {
   int start;
   int val;

   public ByTwos() {
      start = 0;
      val = 0;
   }

   public int getNext() {
      val += 2;
      return val;
   }

   public void reset() {
      start = 0;
      val = 0;
   }
```

1. Интерфейс определяет то, что должен выполнять класс, но не указывает конкретный алгоритм действий.

2. Членами интерфейса могут быть методы, свойства, индексаторы и события.

3. Для реализации интерфейса после имени класса укажите ключевое слово `interface`, затем выполните реализацию для каждого члена интерфейса.

```
  public void setStart(int x) {
    start = x;
    val = x;
  }
}

// Реализация интерфейса Series другим способом.
class ByThrees : Series {
  int start;
  int val;

  public ByThrees() {
    start = 0;
    val = 0;
  }

  public int getNext() {
    val += 3;
    return val;
  }

  public void reset() {
    start = 0;
    val = 0;
  }

  public void setStart(int x) {
    start = x;
    val = x;
  }
}

class SeriesDemo2 {
  public static void Main() {
    ByTwos twoOb = new ByTwos();
    ByThrees threeOb = new ByThrees();
    Series ob;

    for(int i=0; i < 5; i++) {
      ob = twoOb;
      Console.WriteLine("Следующее значение ByTwos " +
                        ob.getNext());
      ob = threeob;
      Console.WriteLine("Следующее значение ByThrees " +
                        ob.getNext());
    }
  }
}
```

В методе `Main()` объект `ob` объявлен в качестве ссылки на интерфейс `series`. Это означает, что он может использоваться с целью хранения ссылок на любой объект. Реализующий интерфейс `Series`. В данном случае он используется для ссылки на объекты `twoOb` и `ThreeOb`, которые являются объектами типа `ByTwos` и `ByThrees` соответственно, и оба они реализуют интерфейс `Series`. Интерфейс ссылочной переменной знает лишь о методах, объявленных в его разделе и указанных его ключевым словом `interface`. В результате объект `ob` не может использоваться при Доступе к любым другим переменным или методам, которые поддерживаются объектом.

Проект 9.1. Создание интерфейса Queue

IcharQ.cs, IQDemo.cs

Для того чтобы увидеть интерфейсы в действии, обратимся к практическому примеру. В предыдущих главах был разработан класс Queue, с помощью которого была реализована одиночная очередь для символов фиксированного размера. Однако реализовать очередь можно самыми различными способами. Например, очередь может иметь фиксированный размер, либо увеличиваться. Очередь может быть *линейной* (в этом случае элементы находятся в очереди постоянно), либо *круговой* (в этом случае элементы могут быть помещены в очередь по мере удаления из нее старых элементов). Очередь также может помещаться в массиве, в связанном списке, в двоичном дереве и других подобных объектах. Каков бы ни был способ реализации очереди, ее интерфейс остается одним и тем же, а методы put() и get() определяют интерфейс независимо от деталей реализации. Поскольку интерфейс очереди отделен от ее реализации, довольно просто можно задать его, оставив на долю каждой реализации определение той или иной специфической черты.

В ходе осуществления данного проекта мы создадим интерфейс очереди символов, а также три его реализации. Во всех трех реализациях для хранения символов используются массивы. Одна очередь будет линейной с фиксированным размером. В качестве второй очереди выступает круговая очередь (в этом случае по достижении конца базового массива методы get и set автоматически возвращаются к началу цикла). Таким образом, в круговой очереди может сохраняться любое количество элементов до тех пор, пока возможно удаление из очереди ранее введенных в нее элементов. В последней реализации создается динамическая очередь, которая в случае превышения изначального размера может автоматически увеличиваться.

Пошаговая инструкция

1. Создайте класс ICharQ.cs и поместите его описание в следующее определение интерфейса:

```
// Интерфейс очереди символов.
public interface ICharQ {
  // Помещение символа в очередь.
  void put(char ch);

  // Выборка символа из очереди.
  char get();
}
```

Как видите, данный интерфейс является достаточно простым и включает лишь два метода. Каждый класс, реализующий ICharQ, нуждается в реализации упомянутых двух методов.

2. Создайте файл IQDemo.cs.
3. Начните создавать проект IQDemo.cs, воспользовавшись указанным ниже кодом класса FixedQueue:

```
/*
   Проект 9.1

   Демонстрация возможностей интерфейса ICharQ.
*/
```

```
using System;

 // Класс очереди фиксированного размера, предназначенный для выполнении
   // операций с символами.
 class FixedQueue : ICharQ {
  char[] q; // Массив, предназначенный для хранении очереди
  int putloc, qetloc; // Индикаторы put и get

  // Конструирование пустой очереди, имеющей, заданный размер.
  public FixedQueue(int size)
    q = new char[size+1]; // Выделение памяти дли очереди.
    putloc = getloc = 0;


  // Помещение символа в очередь.
  public void put(char oh) {
    if (putloc==q.Length-1) {
      Console.WriteLine(" - Очередь заполнена.");
      return;
     }

    putloc++;
    q[putloc] = ch;
  }

  // Получение символа из очереди.
  public char get() {
    if(getloc == putloc) {
      Console.WriteLine(" - Очередь пуста.");
      return (char) 0;
     }

    getloc++;
    return q[getloc];
  }
```

Данная реализация интерфейса `ICharQ` является адаптированной версией класса `Queue`, рассматриваемого в главе 5, и поэтому должна быть вам знакома.

4. В файл `IQDemo.cs` добавьте приведенный ниже код класса `CircularQueue`. Этот класс реализует круговую очередь для символов.

```
// Круговая очередь.
class CircularQueue : ICharQ {
  char[] q; // Очередь содержится в этом массиве.
  int putloc, getloc; // Индикаторы put и get.

  // Пустая очередь заданного размера.
  public CircularQueue (int size) {
    q = new char[size+1]; // Выделение памяти для очереди.
    putloc = getloc = 0;
  }

  // Помещение символа в очередь.
  public void put(char ch) {
    /* Очередь заполнена, либо если putloc меньше чем
       getloc, либо если putloc представляет конец массива,
       а getloc представляет начало массива. */
```

```
    if (putloc+1==getloc |
        ((putloc==q.Length-1) & (getloc==0))) {
      Console.WnteLine(" - Очередь заполнена.");
      return;
    }

    putloc++;
    if(putloc==q.Length) putloc = 0; // Обратный цикл.
    q[putloc] = ch;
  }

  // Выборка символа из очереди.
  public char get() {
    if(getloc == putloc) {
      Console.WriteLine(" - Очередь пуста.");
      return (char) 0;
    }

    getloc++;
    if(getloc==q.Length) getloc = 0; // Обратный цикл.
    return q[getloc];
  }
}
```

При использовании круговой очереди может повторно задействоваться пространство массива, освободившееся после выборки элементов. В результате очередь подобного типа может последовательно включать неограниченное количество элементов, если происходит удаление из нее ранее помещенных элементов. Концептуально все достаточно просто — нужно просто переустановить значение соответствующего индекса в ноль после достижения конца массива — но при этом на первых порах остаются трудности с определением граничных условий. В процессе использования круговой очереди момент ее заполнения определяется не тогда, когда достигается коней базового массива, а в том случае, если при сохранении элемента происходит переписывание невыбранного элемента. При этом метод `put()` должен проверить несколько условий с целью определения того, является ли очередь заполненной. Как и предполагается в комментариях, очередь считается заполненной, если `putloc` будет меньшим `getloc` либо если `putloc` находится в конце массива, а `getloc` — в начале массива. Таким образом, очередь будет пустой, если `putloc` и `getloc` являются эквивалентными.

5. И наконец, в файл `IQDemo.cs` поместите указанный ниже код класса `DynQueue`. При этом реализуется возрастающая очередь, размер которой увеличивается при необходимости.

```
// Динамическая очередь.
class DynQueue : ICharQ {
  char[] q; // В этом массиве хранится очередь
  int putloc, getloc; // the put and get indices

  // Конструирование пустой очереди заданного размера.
  public DynQueue(int size) {
    q = new char[size+1]; // Выделение памяти для очереди.
    putloc = getloc = 0;
  }

  // Помещение символа в очередь.
  public void put(char ch) {
```

```
    if(putloc==q.Length-1)  {
       // Увеличение размера очереди.
       char[] t = new char[q.Length * 2];

       // Копирование элементов в новую очередь.
       for (int. i=0; i < q.Length; i++)
         t[i] = q[i];

       q = t;
     }

    putloc++;
    q[putloc] = ch;


  // Выборка символа из очереди,
  public char get() {
    if(getloc == putloc) {
      Console.WriteLine(" - Очередь пуста.");
      return (char) 0;
    }

    getloc++;
    return q[getloc];
  }
}
```

При использовании этой реализации очереди в случае ее заполнения и попытки сохранения очередного элемента происходит создание нового базового массива, размер которого в два раза больше исходного. Содержимое текущей очереди копируется в массив, а затем ссылка на новый массив присваивается переменной `q`.

6. Для демонстрации возможностей трех реализаций интерфейса `ICharQ` в файл `IQDemo.cs` введите код следующего класса. При этом для доступа ко всем трем очередям используется ссылка интерфейса `ICharQ`.

```
// Демонстрация возможностей очередей.
class IQDemo {
  public static void Main() {
    FixedQueue q1 = new FixedQueue(10);
    DynQueue q2 = new DynQueue(5);
    CircularQueue q3 = new CircularQueue(10);

    ICharQ iQ;

    char ch;
    int i;

    iQ = q1;
    // Помещение нескольких символов в фиксированную очередь.
    for(i=0; i < 10; i++)
      iQ.put((char) ('A' + i));

    // Отображение очереди.
    Console.Write("Содержимое фиксированной очереди: ");
    for(i=0; i < 10; i++) {
      ch = iQ.get () ;
      Console.Write(ch);
```

```
    }
    Console.WriteLine();

    iQ = q2;
    // Помещение некоторых символов в динамическую очередь.
    for(i=0; i < 10; i++)
      iQ.put((char) ('Z' - i));

    // Отображение очереди.
    Console.Write("Содержимое динамической очереди: ");
    for(i=0; i < 10; i++) {
      ch = iQ.qet();
      Console.Write(ch);
    }

    Console.WriteLine();

    iQ = q3;
    // Помещение некоторых символов в круговую очередь.
    for(i=0; 1 < 10; i++)
      IQ.put((char) ('A' + i));

    // Отображение очереди.
    Console.Write("Содержимое круговой очереди: ");
    for(1=0; i < 10; i++) {
      ch = iQ.get();
      Console.Write(ch);
    }

    Console.WriteLine();

    // Помещение дополнительных символов в круговую очередь.
    for(i=10; i < 20; i++)
      iQ.put((char) ('A' + i));

    // Отображение очереди.
    Console.Write("Содержимое круговой очереди: ");
    for(i=0; i < 10; i++) {
      ch = iQ.qet();
      Console.Write(ch);
    }

    Console.WriteLine("\nСохранение и использование" +
                          " круговой очереди. ");

    // Сохранение и использование круговой очереди.
    fcr(i=0; 1 < 20; i++) {
      iQ.put((char) ('A' + i));
      ch = iQ.get();
      Console.Write(ch);
    }

  }
}
```

7. Скомпилируйте программу, задав включение в нее файлов `ICharQ.cs` и `IQDemo.cs`.

8. Ниже приводится результат выполнения программы:

```
Содержимое фиксированной очереди: ABCDEFGHIJ
Содержимое динамической очереди: ZYXWVUTSRQ
Содержимое круговой очереди: ABCDEFGHIJ
Содержимое круговой очереди: KLMNOPQRST
Сохранение и использование круговой очереди.
ABCDEFGHIJKLMNOIPQRST
```

9. А сейчас будут продемонстрированы некоторые вещи, которые вы можете выполнять самостоятельно. В интерфейс `ICharQ` добавьте метод `reset()`, с помощью которого переустанавливается очередь. Создайте метод `static`, посредством которого копируется содержимое очереди одного типа в очередь другого типа.

Свойства интерфейса

Как и в случае с методами, при определении свойств в интерфейсе не указывается тело. Ниже приводится общая форма спецификации свойства:

```
// Свойство интерфейса
type name {
  get;
  set;
}
```

Конечно, `get` и `set` представляют свойства только для чтения и свойства для записи, соответственно.

Ниже приводится продукт переработки интерфейса `Series`, а также код класса `ByTwos`, который использует свойство для получения и задания следующего элемента набора:

```
// Использование свойства в интерфейсе.
using System;

public interface Series {
     // Свойство интерфейса
  int next {
    get; // Возврат следующего числа в наборе,
    set; // Установка следующего числа.
   }
}

// Реализация интерфейса Series.
class ByTwos : Series {
      int val;

  public ByTwos() {
    val = 0;
   }

   // Получение или установка значения.
  public int next {
    get {
      val += 2;
      return val;
    }
```

Объявление свойства в интерфейсе `Series`.

Реализация свойства.

```
    set {
      val = value;
    }
  }
}

// Демонстрация свойства интерфейса.
class SeriesDemo3 {
  public static void Main() {
    ByTwos ob = new ByTwos();

    // Доступ к набору с помощью интерфейса.
    for(int i=0; i < 5; i++)
      Console.WriteLine("Следующее значение " + ob.next);

    Console.WriteLine("\nНачало с 21");
    ob.next = 21;
    for(int i=0; i < 5; i++)
      Console.WriteLine("Следующее значение " + ob.next);
  }
}
```

Результат выполнения программы:

```
Следующее значение 2
Следующее значение 4
Следующее значение 6
Следующее значение 8
Следующее значение 10

Начало с 21
Следующее значение 23
Следующее значение 25
Следующее значение 27
Следующее значение 29
Следующее значение 31
```

Интерфейсные индексаторы

Обратите внимание на общую форму индексатора, объявленного в интерфейсе:

```
// Индексатор интерфейса
element-type this[int index] {
  get;
  set;
}
```

Как и ранее, лишь `get` и `set` представляют собой индексаторы, предназначенные только для чтения и только для записи, соответственно.

Ниже приводится еще одна версия интерфейса `Series`; в ней добавляется индексатор, предназначенный только для чтения и возвращающий i-й элемент набора.

```
// Добавление индексатора в интерфейс.
using System;

public interface Series {
  // Свойство интерфейса
```

```
  int next {
    get; // Возврат следующего числа в наборе.
    set; // Установка следующего числа.
  }

  // Индексатор интерфейса.
  int this [int index] {
    get; // Возврат указанного числа в наборе.
  }
}

// реализация интерфейса Series.
class ByTwos : Series {
  int val;

  public ByTwos() {
    val = 0;
  }

  // Чтение либо задание свойства.
  public int next {
    get {
      val += 2;
      return val;
    }
    set {
      val = value;
    }
  }

  // Получение свойства с помощью индекса.
  public int this[int index] {
    get {
      val = 0;
      for(int i=0; i < index; i++)
        val += 2;
      return val;
    }
  }
}

// Демонстрация индексатора интерфейса.
class SeriesDemo4 {
  public static void Main() {
    ByTwos ob = new ByTwos();

    // Доступ к набору с помощью свойства
    for(int i=0; i < 5; i++)
      Console.WriteLine("Следующее значение " + ob.next);

    Console.WriteLine("\nНачало с 21");
    ob.next = 21;
    for(int i=0; i < 5; i++)
      Console.WriteLine("Следующее значение " +
                        ob.next);
```

Указание индексатора, предназначенного только для чтения.

Реализация индексатора.

```
    Console.WnieLine("\nПереустановка в 0");
    ob.next = 0;

    // Доступ к набору с помощью индексатора
    for(int i=0; 1 < 5; i++)
      Console.WriteLine("Следующее значение " + ob.next);
  }
}
```

Результат выполнения программы:

```
Следующее значение 2
Следующее значение 4
Следующее значение 6
Следующее значение 8
Следующее значение 10

Начало с 21
Следующее значение 23
Следующее значение 25
Следующее значение 27
Следующее значение 29
Следующее значение 31

Переустановка в 0
Следующее значение 0
Следующее значение 2
Следующее значение 4
Следующее значение 6
Следующее значение 8
```

Минутный практикум

1. Может ли переменная ссылки интерфейса указывать на объект, который реализует этот интерфейс?
2. Может ли свойство интерфейса быть предназначенным только для чтения?
3. Имеют ли тела индексатор и свойства, определенные внутри интерфейса?

Наследование интерфейсов

Возможно наследование интерфейсов. В этом случае используется синтаксис, весьма напоминающий синтаксис наследования классов. Если класс реализует интерфейс, который наследует другой интерфейс, должна обеспечиваться реализация для всех членов, определенных и составе цепи наследования интерфейсов. Обратите внимание на следующий пример:

```
// Один интерфейс может наследовать другой интерфейс,
using System;

public interface A {
```

1. Да.
2. Да.
3. Нет.

```
 void meth1();
 void meth2();
 }

   // Интерфейс B включает методы meth1() и meth2() - он добавляет meth3().
    public interface B : A {
   void meth3();
 }

// Этот класс должен реализовывать все методы A и B
 class MyClass : B {
  public void meth1() {
    Console.WriteLine("Реализация meth1().");
 }

  public void meth2() {
    Console.WriteLine("Реализация meth2().");
  }

  public void meth3() {
    Console.WriteLine("Реализация meth3().");
   }
}

class IFExtend {
  public static void Main() {
    MyClass ob = new MyClass();

    ob.meth1();
    ob.meth2();
    ob.meth3();
   }
}
```

B наследует A.

Ответы профессионала

Вопрос. Если один интерфейс наследует другой интерфейс, можно ли объявить член наследующего интерфейса, скрывающего член, определенный базовым интерфейсом?

Ответ. Да. Если сигнатура члена наследующего интерфейса совпадает с сигнатурой одного из наследуемых интерфейсов, имя наследуемого интерфейса будет скрыто. Как и в случае с наследованием класса, подобное сокрытие приведет к отображению предупреждающего сообщения независимо от того, будет ли указан член наследующего интерфейса с помощью ключевого слова `new`.

В качестве эксперимента, можно попытаться удалить реализацию метода `meth1()` в классе `MyClass`. Это приведет к появлению ошибки компиляции. Как упоминалось ранее, любой класс, реализующий интерфейс, должен реализовывать все методы, которые определены с помощью данного интерфейса, включая те из них, которые наследуются из других интерфейсов.

Явная реализация

При реализации члена интерфейса можно *полностью квалифицировать* его имя в качестве имени интерфейса. В результате создается *явная реализация члена интерфейса,* либо, сокращенно, *явная реализация.* Например, следующий код

```
Interface IMyIF {
  int myMeth(int x);
}
```

допустим при реализации `IMyIF`, как показано ниже:

```
class MyClass : ImyIF {
  int  ImyIF.myMeth(int  x)  {
    return x / 3;
  }
}
```

Полностью определенное имя используется при создании явной реализации.

Как видите, при реализации члена `myMeth()` класса `IMyIF` указывается его полное имя, включая имя его интерфейса.

Необходимость явной реализации для члена интерфейса может обосновываться двумя причинами. Во-первых, класс может реализовывать два интерфейса, причем в обоих объявляются методы с помощью одного и того же имени и типа сигнатуры. Благодаря полному определению имен устраняется двусмысленность подобной ситуации. Во-вторых, при реализации метода посредством его полностью определенного имени определяется объем частной реализации, который не доступен для кода, не относящегося к классу. А сейчас перейдем к рассмотрению практических примеров.

Следующая программа содержит интерфейс `IEven`, в котором определяются два метода: `isEven()` и `isOdd()`. Эти методы используются для определения того, является число четным либо нечетным. Затем класс `MyClass` реализует интерфейс `IEven`. После этого явно реализуется метод `isOdd()`.

```
// Явная реализация члена интерфейса.
using System;

interface IEven {
  bool isOdd(int x);
  bool isEven(int x);
}

class MyClass : IEven {
  // Явная реализация,
  bool IEven.isodd(int x) {
    if((x%2) != 0) return true;
    else return false;
  }

  // Обычная реализация.
  public bool isEven(int x) {
    IEven o = this; // Ссылка на вызывающий объект.

    return !o.isOdd(x);
  }
}
```

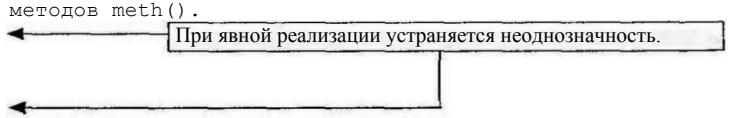
Явно реализован метод `isOdd()`. В результате он будет действительно частным.

```
class Demo {
 public static void Main() {
   MyClass ob = new MyClass();
   bool result;

   result = ob.isEven(4);
   if(result) Console.WriteLine("4 четно.");
   else Console.WriteLine("3 нечетно.");

       // result = ob.isOdd(); // Ошибка, не открыто.
  }
}
```

Поскольку метод `Odd()` реализован явно, он недоступен вне класса `MyClass`. Благодаря этому реализация станет по-настоящему эффективной. Метод `isOdd()` внутри класса `MyClass` может стать доступным только после того, как на него будет сделана ссылка из интерфейса. Причина этого заключается в том, что он вызывается с помощью реализации о в методе `isEven()`.

Ниже приводится пример, в котором два интерфейса реализуются и объявляются с помощью метода `meth()`. С целью преодоления неоднозначности, возникающей в подобной ситуации, используется явная реализация.

```
// Использование явной реализации с целью устранения неоднозначности.
using System;

interface IMyIF_A {
 int meth(int x);
}

interface IMyIF_B {
 int meth(int x);
}

 // В классе MyClass реализуются оба интерфейса.
class MyClass : IMyIF_A, IMyIF_B {
 IMyIF_A a_ob;
 IMyIF_B b_ob ;

    // Явная реализация двух методов meth().
 int IMyIF_A.meth(int x) {
   return x + x;
 }
 int IMyIF_B.meth(int x) {
   return x * x;
 }

    // Вызов метода meth() с помощью ссылки интерфейса.
 public int methA(int x) {
   a_ob = this;
   return a_ob.aeth(x); // Вызов IMyIF_A
 }

 public int methB(int x) {
   b_ob = this;
   return b_ob.meth(x); // Вызов IMyIF_B
 }
```



```
class FQIFNames {
  public static void Main() {
    MyClass ob = new MyClass();

    Console.Write("Вызов метода IMyIF_A.meth(): ");
    Console.WriteLine(ob.methA(3));

    Console.Write("Вызов метода IMyIF_B.meth(): ");
    Console.WriteLine(ob.methB(3));
  }
}
```

Результат выполнения программы:

```
Вызов метода IMyIF_A.meth(): 6
Вызов метода IMylF_B.meth(): 9
```

При просмотре кода программы обратите внимание на то, что метод `meth()` имеют одни и те же сигнатуры в `IMyIF_A` и в `IMyIF_B`. Таким образом, в случае, если `MyClass` реализует оба этих интерфейса, он должен явно реализовать каждый из них по отдельности, полностью квалифицируя их имена в ходе осуществления процесса. Поскольку единственным способом, посредством которого может быть вызван явно реализованный метод, является ссылка интерфейса, метод `MyClass` создает для подобных ссылки, одна из которых предназначена для `IMyIF_A`, а вторая — для `IMyIF_B`. Затем вызываются оба собственных метода класса с помощью методов интерфейса, в результате чего устраняется неоднозначность.

Минутный практикум

1. Если интерфейс A наследуется интерфейсом B, а интерфейс B наследуется классом C, то может ли класс C реализовывать все члены классов A и B либо только те из них, которые определены классом B?
2. Назовите две причины, в силу которых может потребоваться явная реализация интерфейса.

Структуры

Как вы знаете, классы представляют собой ссылочные типы. Это означает, что доступ к объектам классов производится с помощью ссылок. Это в корне отличается от типов значений, доступ к которым осуществляется непосредственно. Однако иногда бывает полезно обеспечить непосредственный доступ к объекту в силу соображений эффективности. Доступ к объектам класса с помощью ссылок влечет дополнительные накладные расходы при каждом сеансе доступа. При этом также потребляется дополнительное пространство на жестком диске. При работе с очень маленькими объектами величина дополнительного потребляемого пространства может иметь большое значение. С целью решения подобных проблем в C# предлагаются *структуры*. Структура подобна классу, но она имеет скорее тип значения, но никак не тип ссылки.

1. Реализующий класс должен включать все члены, указанные в иерархии интерфейсов.

2. Явная реализация может использоваться для устранения неоднозначности, а также для препятствия отображению члена интерфейса.

Объявление структур выполняется с помощью ключевого слова `struct`. Так следует отметить, что структуры синтаксически подобны классам. Ниже приводит Обобшснnаи форма объекта `struct`:

```
struct name : interfaces {
  // Объявлений членов
}
```

Имя структуры указывается с помощью параметра `name`.

Структуры не могут наследовать другие структуры либо классы, а также не могут применяться в качестве основы для других структур либо классов. Однако каждая структура может реализовывать один или больше интерфейсов. Реализуемые интерфейсы указываются после имени структуры в виде списка, разделенного запятыми.

Как и в случае с классами, члены структуры включают методы, поля, индексаторы свойства, операторные методы, а также события. Кроме того, структуры могут определять конструкторы, но не деструкторы. Однако для структуры не может быть определен конструктор, заданный по умолчанию (без параметров). Причина этому заключается в том, что конструктор, заданный по умолчанию, автоматически определяется для всех структур и не может быть изменен.

Объект структуры может быть создан с помощью ключевого слова `new` точно так же как объект класса (но не обязательно). Как только будет использовано ключевое слово `new`, вызывается указанный конструктор. Если названное ключевое слово не применяется, объект по-прежнему создастся, но не инициализируется. Благодаря этом не требуется выполнять инициализацию вручную.

Ниже приводится пример использования структуры:

```
// Демонстрация возможностей структуры.
using System;

// Определение структуры.
struct account. {
  public string name;
  public double balance;

  Public account(string n, double b) {
    name = n;
    balance = b;
  }
}

// Демонстрация возможностей структуры account.
class StructDemo {
  public static void Main() {
    account acc1 - new account("Том", 1232.22); // Явный конструктор.
    account acc2 — new account(); // Конструктор, заданный по умолчанию.
    account acc3; // Конструктор отсутствует.

    Console.WriteLine(accl.name + " имеет на счету " + acc1.balance);
    Console.WriteLine();

    if(acc2.narae == null) Console.WriteLine("acc2.name равно null.");
    Console.WriteLine("acc2.balance равен " + acc2.balance);
    Console.WriteLine();
```

Определение структуры.

```
    // Должен инициализировать acc3 до его использования.
    acc3.name = "Мэри";
    acc3.balance = 99.33;
    Console.WriteLine(acc3.name + " имеет баланс " + acc3.balance);
  }
}
```

Результат выполнения программы:

```
Том имеет на счету 1232.22

acc2.name равно null.
acc2.balance равен 0

Мэри имеет на счету 99.33
```

Ответы профессионала

Вопрос. Известно, что в C++ также имеются структуры и использует ключевое слово `struct`. Подобны ли структуры языков программирования C# в C++?

Ответ. Нет. В C++ ключевое слово `struct` определяет тип класса. Таким образом, в C++ ключевые слова `struct` и `class` практически эквивалентны. (Разница заключается в способе реализации доступа к членам класса, заданного по умолчанию.) В C# `struct` определяет тип значения, а `class` определяет тип ссылки.

В примере программы структура инициализируется либо посредством ключевого слова `new`, используемого для вызова конструктора, либо посредством простого объявления объекта. Если используется ключевое слово `new`, выполняется инициализация полей структуры. Процесс инициализации происходит либо с помощью конструктора, заданного по умолчанию, который инициализирует все поля с помощью стандартных значений, либо путем использования конструктора, определенным пользователем. Если ключевое слово `new` не применяется, объект не инициализируется, а его полям должны быть назначены значения до использования объекта.

Перечисления

Под *перечислением* понимается набор именованных целочисленных констант, определяющих все допустимые значения переменной для данного типа. Подобный у объектов весьма распространен. Ниже приводится перечисление, включающее описание монет США:

```
penny, nickel, dime, quarter, half-dollar, dollar
```

Ключевое слово `enum` определяет перечислимый тип. Рассмотрим общую форму используемую при определении перечисления:

```
enum name { enumeration list };
```

Имя типа, используемого в перечислении, указывается с помощью параметра `name`. Список `enumeration list` представляет собой список идентификаторов, разделенных запятыми.

В следующем фрагменте кода определяется перечисление coin:

```
enum coin { penny, nickel, dime, quarter, half-dollar, dollar};
```

Важным в понятии «перечисление» является тот факт, что каждому из символов соответствует целочисленное значение. В результате область применения символов совпадает с областью использования целочисленных значений. Целочисленное значение для каждого символа больше значения предшествующего символа. По умолчанию значение первого символа перечисления равно нулю.

Доступ к элементам перечисления осуществляется посредством имен типов, с использованием точечного оператора. Обратите внимание на код

```
Console.WriteLine("penny is " + coin, penny +
                    " nickel is " + coin.nickel);
```

который отображает следующее сообщение:

```
penny is 0 nickel is 1
```

Ниже приводится пример программы, демонстрирующей возможности перечисления coin:

```
// Демонстрация возможностей перечисления
using System;

 class EnumDemo {
  enum coin { penny, nickel, dime, quarter,
              half_dollar, dollar };

  public static void Main() {
    string[] names = {
      "penny",
      "nickel",
      "dime",
      "quarter",
      "half_dollar",
      "dollar"
    };

    coin i; // Объявление переменной для обработки перечисления

    // Использование переменной i для цикла
    // обработки перечисления
    for(i — coin.penny; i <= coin.dollar; i++)
      Console.WriteLine(names[(int)i] + " имеет значение " + i);

  }
}
```

Переменная coin может контролировать цикл for

Результат выполнения программы:

```
      penny имеет значение 0
          nickel имеет значение 1
dime имеет  значение 2
quarter имеет значение 3
half_dollar имеет значение 4
     dollar имеет значение 5
```

Обратите внимание на то. что управление циклом for осуществляется с помощью переменной, имеющий тип coin. Поскольку перечисление имеет целочисленный

тип, значение перечисления может присваиваться там же, где могут назначаться целочисленные значения. Однако при использовании значения перечисления целью индексации массива `names` требуется провести преобразование типа. С другой стороны, как только значения перечислимых значений в `coin` начинаются с нулем эти значения могут использоваться при индексировании массива `names` с целью получения названий монет. Техника, заключающаяся в сопоставлении осмысления строк и значений перечисления, является весьма полезной.

Инициализация перечисления

С помощью инициализатора можно указать значения одного или большего количества символов. Для этого после каждого символа должен следовать знак равенства и целочисленное значение. Символы, отображаемые после инициализаторов, представляют собой назначенные значения, которые должны быть больше, чем предыдущее значение инициализации. Например, в следующем коде значение 100 присваивается символу `quarter`:

```
enure coin { penny, nickel, dime, quarter-100,
             half_dollar, dollar};
```

Теперь символы имеют такие значения:

```
penny       0
nickel      1
dime        2
quarter     100
half_dollar 101
dollar      102
```

Указание базового типа перечисления

По умолчанию перечисления основываются на типе `int`, но можно создавать перечисления любого целочисленного типа, за исключением `char`. Для указания типа, отличного от `int`, поместите наименование базового типа после имени перечисления, разделив их между собой двоеточием. Например, после применения следующего оператора coin становится перечислением, основанным на типе `byte`.

```
enum coin : byte { penny, nickel, dime, quarter,
             haif_dollar, dollar};
```

Теперь, например, значение, присваиваемое члену `coin.penny`, имеет тип `byte`.

Минутный практикум

1. Нужно ли при создании структуры пользоваться ключевым словом `new`?
2. Может ли для структуры определяться деструктор?
3. Что такое перечисление?

1. Нет, объект `struct` может создаваться подобно объекту любого другого типа без использования ключевого слова new. Однако при этом объект не будет инициализирован.

2. Нет, структуры не могут обладать деструкторами.

3. Перечисление представляет собой перечень именованных целочисленных констант.

Контрольные вопросы

1. Ключевой афоризм для C# звучит так: «Один интерфейс, много методов». Какие свойства лучше всего иллюстрируют это высказывание?
2. Сколько классов может реализовать интерфейс? Сколько интерфейсов может реализовать класс?
3. Могут ли наследоваться интерфейсы?
4. Должен ли класс реализовывать вес члены интерфейса?
5. Может ли интерфейс объявлять конструктор?
6. Создайте интерфейс для класса `Venicle` (глава 7). Назначьте ему имя `IVenicle`.
7. Создайте интерфейс для безопасных массивов. Добейтесь этого путем адаптации массива, устойчивого к отказам, из последнего примера главы 7.
8. Каковы различия между `struct` и `class`?
9. Покажите, каким образом можно создать перечисление для планет солнечной системы. Назовите его `Planets`.

Обработка исключений

- Использование блоков try и catch
- Если исключение не перехвачено...
- Использование нескольких операторов catch
- Вложенные блоки try
- Генерирование исключений
- Объект Exception
- Использование ключевого слова finally
- Исследование встроенных исключений C#
- Создание собственных классов исключений
- Использование ключевых слов checked и unchecked

В главе рассматривается обработка исключений. *Исключение* представляет собой ошибку, происходящую во время выполнения программы. С помощью подсистемы обработки исключений для C# можно обрабатывать такие ошибки, не вызывая краха программы. Решение этой задачи в C# основано на усовершенствованных методиках, применяемых в языках программирования C++ и Java. Поэтому для читателей, знакомых с основами одного из этих языков, эти методики уже известны. Однако обработка исключений в C# отличается четкостью и простотой реализации.

Наличие механизма обработки исключений позволяет упростить процесс написания кода. Ранее код обработки исключений полностью создавался программистом. Например, если язык программирования не имеет средств обработки исключений, то необходимо предусмотреть возврат методами кодов ошибок, возникающих в результате сбоев программы, и обработку этих кодов. Подобный подход неудобен и чреват возникновением новых ошибок. Обработка исключений ускоряет процесс обработки ошибок, поскольку программа может содержать блок кода, именуемый *обработчиком исключении.* Этот блок кода будет выполняться автоматически в случае возникновения каких-либо ошибок. Необязательно вручную проверять правильность выполнения каждой отдельной операции или вызова метода. При появлении ошибки происходит ее обработка с помощью обработчика ошибок.

Другая причина, по которой важна обработка исключений, состоит в том, что в C# определены стандартные исключения для наиболее часто встречающихся ошибок, таких как деление на ноль или выход значения индекса за пределы диапазона. Поэтому для устранения подобных ошибок в программе нужно только следить за возникновением соответствующих исключений и обрабатывать их.

Таким образом, для успешного овладения языком программирования C# необходимо в совершенстве освоить методику работы с подсистемой C#, выполняющей обработку исключений.

Класс `System.Exception`

В языке C# исключения представлены классами. Все классы исключений могут быть Унаследованы от встроенного класса исключений `Exception`, являющегося частью пространства имен `System`. Поэтому все исключения представляют собой подклассы класса `Exception`.

Из класса `Exception` наследуются классы исключений `SystemException` и `ApplicationException`. Таким образом, в C# определяются две общие категории исключений: образованные средой исполнения C# (то есть CLR) и сгенерированные программными приложениями. Ни `SystemException`, ни `ApplicationException` не добавляют ничего нового в класс `Exception`. Эти классы фактически определяют верхние точки двух различных иерархий исключений.

В C# определены несколько встроенных исключений, наследуемых из класса `SystemException`. Например, если возникает ситуация с делением на ноль, генерируется исключение `DivideByZeroException`. Как будет отмечено далее в этой главе, на основе класса исключений `ApplicationException` пользователь может сформировать свои собственные классы исключений.

Основы обработки исключений

Обработка исключений в C# выполняется с применением четырех ключевых слов: try, catch, throw и finally. Эти ключевые слова образуют взаимосвязанную подсистему, в которой использование одного из ключевых слов влечет за собой использование других. Все упомянутые ключевые слова будут подробнее рассмотрены ниже. Однако сначала полезно получить общее представление о той роли, которая отведена при обработке исключений каждому из этих ключевых слов. Рассмотрим вкратце функции каждого из них.

В блок try помещаются операторы программы, за выполнением которых необходимо следить на предмет возникновения исключений. Возникающее исключение перехватывается и обрабатывается блоком catch. Для генерирования исключения применяется ключевое слово throw. В блок finally помещается код, который всегда выполняется при выходе из блока try.

Использование блоков try и catch

Обработка исключений основана на использовании блоков try и catch. Эти блоки работают совместно: нельзя задать блок try и не задать catch, и наоборот. Ниже, приводится общий синтаксис блоков try/catch, выполняющих обработку исключений:

```
try {
   // Блок кода, для которого выполняется мониторинг ошибок.
}

catch (ExcepType1 exOb) {
   // Обработчик исключений ExcepType1.
}
catch (ExcepType2 exOb) {
   // Обработчик исключений ExcepType2.
}
```

Параметры ExcepType задают типы обрабатываемых исключений. После генерирования исключения происходит его перехват соответствующим оператором catch, выполняющим обработку этого исключения. Как показано в общей форме синтаксиса, с блоком try может ассоциироваться более одного оператора catch. Тип исключения определяет выполняемый оператор catch. Если возникшее исключение соответствует типу исключения, указанному в операторе catch, то выполняется блок команд этого оператора (а все остальные блоки catch игнорируются). При перехвате исключений оператором catch соответствующий объект исключения exOb получает его значение.

Фактически указание объекта exOb не является обязательным. Если обработчик исключений не нуждается в доступе к объекту исключения (это часто бывает на практике), нет необходимости в указании exOb. Учитывая сказанное, для многих примеров в главе объекты исключения задавать не надо.

Важно отметить, что если исключение не генерируется, работа операторов блока try завершается в обычном порядке, а все соответствующие ему операторы catch пропускаются. Выполнение программы возобновляется с первого оператора, следующего за последним оператором catch. Таким образом, операторы catch вызываются только в том случае, если генерируется исключение.

Простой пример исключения

Ниже приводится простой пример, демонстрирующий методику отслеживания и перехвата исключений. Вы, наверное, знаете, что при обращении к индексу, выходящему за пределы массива, возникает ошибка. В этом случае система выполнения C# генерирует исключение `IndexOutOfRangeException`, представляющее собой стандартное исключение, определенное в C#. Назначение следующей программы заключается в генерировании и перехвате подобного исключения:

```
// Демонстрация обработки исключений.
using System;

class ExcDemo1 {
  public static void Main() {
    int[] nums = new int[4];

    try {
      Console.Writeiine("Эта строка выводится до генерирования исключения."!;

      // Генерирование исключения вследствие выхода индекса за пределы диапазона.
      nums[7] = 10;
      Console.WriteLine("Этот текст не будет отображен.");
    }
    catch (IndexOutOfRangeException) {
      // Перехват исключения.
      Console.WriteLine("Индекс за пределами диапазона!");
    }
    Console.WriteLine("После оператора catch.")
  }
}
```

Попытка использования индекса массива `nums`, находящегося за пределами диапазона.

Результат выполнения программы:

```
Эта строка выводится до генерирования исключения.
Индекс за пределами диапазона!
После оператора catch.
```

В данной программе кратко рассматриваются некоторые основные моменты, связанные с обработкой исключений. Во-первых, код, который требуется отслеживать на предмет выявления ошибок, заключается в блок `try`. Во-вторых, в случае возникновения исключения (в данном случае — по причине попытки использования значения индекса массива `nums`. выходящего за пределы диапазона) таковое перехватывается оператором `catch`. Начиная с этого момента контроль передается блоку `catch`, а блок `try` завершает свою работу. При этом блок `catch` фактически *не* вызывается. Скорее, этому блоку передается управление. В результате оператор `WriteLine()`, следующий за оператором использования индекса, значение которого выходит за пределы диапазона, выполнен не будет. После выполнения оператора `catch` управление передается операторам, следующим за блоком `catch`. Решение проблем, вызывающих возникновение исключения, — удел обработчика исключений. При этом программа продолжает выполняться и не завершается аварийно.

Обратите внимание на то, что в конструкции `catch` не указан ни один параметр. Как упоминалось ранее, параметр требуется только тогда, когда необходим доступ к объектам исключений. В некоторых случаях значение объекта исключения может быть Использовано обработчиком исключений с целью получения дополнительных сведений

об ошибке, но часто достаточно самого факта возникновения исключения. Таким образом, отсутствие параметра `catch` в обработчике исключений (как в предыдущем примере программы) не является чем-либо необычным.

Ранее уже упоминалось о том, что если в блоке `try` исключение не возникает, то операторы `catch` не вызываются и управление программой передается оператору следующему после оператора `catch`. С целью проверки этого факта в предыдущем примере программы замените строку

```
nums[7]  = 10;
```

строкой

```
nums[0] = 10;
```

В результате исключение не возникнет, а блок `catch` не будет вызван.

Второй пример исключения

Важно понимать, что весь код, входящий в состав блока `try`, контролируется на предмет наличия исключений. При этом указываются исключения, которые могут генерироваться методом, вызываемым внутри блока `try`. Исключения подобного типа могут перехватываться этим же блоком `try` (предполагается, что метод сам по себе не выполняет перехват исключений). Ниже приводится пример корректной программы:

```
/* Исключение может генерироваться с помощью одного
метода, а затем перехватываться другим методом. */
using System;

class ExcTest {
  // Генерирование исключения.
  public static void genException() {
    int[] nums = new int[4];

    Console.WriteLine("Эта строка выводится до генерирования исключения.");

    // Генерирование исключения в случае выхода индекса за пределы диапазона
    nums[7] = 10;
    Console.WriteLine("Этот текст не будет отображен.");
  }
}

class ExcDemo2 {
  public static void Main() {

    try {
      ExcTest.genException();
    }
    catch (IndexOutOfRangeException) {
      // catch the exception
      Console.WriteLine("Индекс за пределами диапазона!");
    }
    Console.WriteLine("После оператора catch.");
  }
}
```

Здесь генерируется исключение.

Здесь перехватывается исключение.

результат выполнения этой программы аналогичен результату выполнения предыдущей:

```
Эта строка выводится до генерирования исключения.
Индекс за пределами диапазона!
После оператора catch.
```

Как только метод `genException()` вызывается внутри блока `try`, генерируемое в результате исключение (если оно до сих пор не перехвачено) перехватывается с помощью оператора `catch`, входящего в состав метода `Main()`. Понятно, что если метод `genException()` перехватит исключение, оно уже не будет возвращено методу `Main()`.

Минутный практикум

1. Что такое исключение?
2. Частью какого оператора должен быть код, отслеживаемый исключениями?
3. Каковы функции оператора `catch`? Что происходит после выполнения этого оператора?

Если исключение не перехвачено

Перехват и обработка одного из стандартных исключений C# (выполняемые в предыдущем примере) предотвращают аварийное завершение программы. После возникновения исключения оно должно быть перехвачено и обработано неким фрагментом кода. Если программа не перехватывает исключение, последнее будет перехвачено системой выполнения C#. Проблема, появляющаяся в этом случае, заключается в том, что система выполнения отображает сообщение об ошибке и завершает выполнение программы. Например, в следующей версии предыдущего примера исключение, возникающее в результате выхода за пределы диапазона, не перехватывается программой:

```
// Позволяет системе выполнения C# обрабатывать ошибки.
using System;

class NotHandled {
  public static void Main() {
    int[] nums = new int[4];

    Console.WriteLine("Эта строка выводится до генерирования исключения.");

    // Генерирование исключения при выходе значения индекса за пределы
    // диапазона.
    nums[7] = 10;
  }
}
```

1. В роли исключения выступает ошибка времени выполнения.

2. Для отслеживания исключений потребуется включить код в состав блока `try`.

3. Оператор `catch` перехватывает исключения. Так как оператор `catch` не вызывается из программы, то после выполнения блока `catch` управление не передается обратно оператору программы, при выполнении которого возникло исключение. Выполнение программы продолжается с операторов, находящихся после блока `catch`.

В этом случае выполнение программы прерывается и отображается следующее сообщение об ошибке:

```
Unhandled Exception: System.IndexOutOfRangeException:
           Exception of type System.IndexOutOfRangeException
           was thrown.
   at NotHandled.Main()
```

Хотя сообщения подобного типа являются весьма полезными в случае отладки, весьма нежелательно, чтобы их видели пользователи данной программы! По этой причине важно, чтобы программа сама обрабатывала исключения.

Ранее уже упоминалось о том, что тип исключения должен соответствовать типу, указанному с помощью оператора `catch`. Если это не так, исключение может быть не перехвачено. Например, в следующей программе предпринимается попытка обработки ошибки, возникающей вследствие выхода за границы массива, путем перехвата оператором `catch` исключения `DivideByZeroException` (еще одно встроенное исключение C#), Как только будут перейдены границы массива, генерируется исключение `IndexOutOfRange`, которое не перехватывается оператором `catch`. В результате происходит аварийное завершение программы.

```
// Эта программа неверно обрабатывает исключение!
 using System;

class ExcTypeMismatch {
  public static void Main() {
    int[] nums = new int[4];

    try {
      Console.WriteLine("Эта строка выводится до генерирования исключения.");

     // Генерирование исключения выхода значения индекса массива за пределы
     // диапазона.
      nums[7] = 10;
      Console.WriteLine("Этот текст не отобразится.");
    }

    /* Невозможно перехватить ошибку выхода за границы массива
       с помощью исключения DivideByZeroException. */
    catch (DivideByZeroException) {
      // Перехват исключения
      Console.WriteLine("Индекс за пределами диапазона!");
    }
    Console.WriteLine("После оператора catch.");
  }
}
```

Здесь генерируется исключение `IndexOutOfRangeException`.

Попытка перехвата ошибки с помощью исключения `DivideByZeroException`.

Результат выполнении программы:

```
Эта строка выводится до генерирования исключения.

 Unhandled Exception: System.IndexOutOfRangeException:
           Exception of type System.IndexOutOfRangeException
           was thrown.
   at ExcTypeMismatch.Main()
```

результаты выполнения программы показывают, что оператор `catch`, рассчитанный на исключение `DivideByZeroException`, не перехватывает исключение `IndexOutOfRangeException`.

Изящная обработка ошибок с помощью исключений

Одним из основных преимуществ обработки исключений является то, что в этом случае программа может реагировать на ошибки, а затем продолжать выполнение. Обратите внимание на следующий пример, в котором элементы одного массива делятся на элементы другого. Если произошло деление на ноль, генерируется исключение `DivideByZeroException`. В программе выполняется обработка исключения с отображением сообщения об ошибке, а затем программа продолжает выполняться. Таким образом, ситуация деления на ноль не приводит к появлению ошибки выполнения с последующим аварийным завершением работы программы. Вместо этого происходит изящная обработка ошибки, в результате которой возможно продолжение выполнения программы.

```
// Изящная обработка ошибок и продолжение выполнения программы.
using System;

class ExcDemo3 {
  public static void Main() (
    int[] numer = { 4, 8, 16, 32, 64, 128 );
    int[] denom = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i<numer.Length; i++) {
      try {
        Console.WriteLine(numer[i] + " / " +
                          denom[i] + " будет " +
                          numer[i]/denom[i]);
      }
      catch (DivideByZeroException) {
        // Перехват исключения
        Console.WriteLine("Нельзя делить на ноль!");
      }
    }
  }
}
```

Вот результат выполнения программы:

```
4 / 2 будет 2
Нельзя делить на ноль!
16 / 4 будет 4
     32 / 4 будет 8
Нельзя делить на ноль!
128 / 8 будет 16
```

Этот пример демонстрирует еще один важный факт: как только исключение обрабатывается, оно устраняется из системы. Таким образом, в программе, в которой блок `try` заключен в тело цикла, будут обрабатываться все исключения. Благодаря этому возможна обработка повторяющихся ошибок.

Минутный практикум

1. Каким должен быть тип исключения в операторе `catch`?
2. Что произойдет в случае, если исключение не будет перехвачено?
3. Что должна выполнять программа при возникновении исключения?

Использование нескольких операторов `catch`

С одним блоком `try` может быть связано более одного блока `catch`. Подобная практика является общепринятой. Однако каждый блок `catch` должен перехватывать исключения различного типа. Приведенная ниже программа перехватывает ошибки, связанные с выходом индекса за границы массива и с делением на ноль.

```
// Использование нескольких операторов catch.
using System;

 class ExcDemo4 {
  public static void Main() {
    // Массив numer содержит больше элементов, чем denom.
    int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i<numer.Length; i++) {
      try {
        Console.WriteLine(numer[i] + " / " +
                          denom[i] + " будет " +
                          numer[i]/denom[i]);
      }
      catch (DivideByZeroException) {
        // Перехват исключения
        Console.WriteLine("Нельзя делить на ноль!");
      }
      catch (IndexOutOfRangeException) {
        // Перехват исключения
        Console.WriteLine("Не найдены совпадающие элементы.");

    }
  }
}
```

Несколько операторов `catch`.

Результат выполнения программы:

```
4 / 2 будет 2
Нельзя делить на ноль!
16 / 4 будет 4
32 / 4 будет 8
Нельзя делить на ноль!
```

1. Тип исключения в операторе `catch` должен соответствовать типу перехватываемого исключения.

2. Неперехваченное исключение непременно приводит к досрочному прекращению выполнения программы.

3. Программа должна осуществлять изящную и рациональную обработку исключений. При этом нельзя оставлять необработанные исключения и следует избегать нестандартных завершений.

```
    128 / 8 будет 16
Не найдены совпадающие элементы.
Не найдены совпадающие элементы.
```

Из результата выполнения программы видно, что каждый оператор `catch` соответствует своему собственному типу исключения.

Как правило, выражения `catch` проверяются в порядке их записи в программе. В итоге выполняется только тот блок, исключение которого совпадает с возникшим. Остальные блоки `catch` будут в этом случае проигнорированы.

Перехват всех исключений

Иногда требуется перехватывать все исключения, не обращая внимания на их тип. Для выполнения этой задачи воспользуйтесь оператором `catch`, для которого не указываются параметры. Такой оператор перехватывает все исключения. В приведенном ниже примере возникающие исключения `IndexOutOfRangeException` и `DivideByZeroException` перехватываются и обрабатываются одним и тем же блоком `catch`:

```
 // Использование оператора catch, "перехватывающего все".
using System;

class ExcDemo5 {
  public static void Main() {
    // Массив numer содержит больше элементов, чем denom.
    int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i<numer.Length; i++) {
      try {
        Console.WriteLine(numer[i] + " / " +
                          denom[i]  + "  будет " +
                          numer[i]/denom[i]);
  }
      catch {
        Console.WriteLine("Возникло исключение.");
       }
    }
  }
}
```

Перехватываются все исключения.

Результат выполнения программы:

```
4 / 2 будет 2
Возникло исключение.
16 / 4 будет 4
32 / 4 будет 8
Возникло исключение.
128 / 8 будет 16
Возникло исключение.
Возникло исключение.
```

Вложенные блоки try

Блоки try могут вкладываться друг в друга. Исключение, генерируемое во внутреннем блоке try и не перехваченное блоком catch, соответствующим данному внутреннему блоку try, распространяется на внешний блок try. Ниже показан пример возникновения исключения IndexOutOfRangeException, которое не перехватывается внутренним блоком try, но зато перехватывается внешним блоком try:

```
// Использование вложенного блока try.
using System;

class NestTrys {
  public static void Main() {
    // Массив numer содержит больше элементов, чем denom.
    int[] numer   = { 4, 8,  16, 32,  64, 128, 256,   512 };
    int[] denom   = { 2,  0,  4,  4,  0,  8 };

    try {  // Внешний блок try
      for(int i=0; i<numer.Length; i++) {
        try { // Вложенный блок try
          Console.WriteLine(numer[i] + " / " +
                            denom[i] + " будет " +
                            numer[i]/denom[i]);
        }
        catch (DivideByZeroException) {
          // Перехват исключения.
          Console.WriteLine("Нельзя делить на ноль!");
        }
      }
    }
    catch (IndexOutOfRangeException) {
      // Перехват исключения.
      Console.WriteLine("Не найдены совпадающие элементы.");
      Console.WriteLine("Критическая ошибка - выполнение программы прервано.");
    }
  }
}
```

Вложенные блоки try.

Результат выполнения программы:

```
4 / 2 будет 2
Нельзя делить на ноль!
16 / 4 будет 4
32 / 4 будет 8
Нельзя делить на ноль!
128 / 8 будет 16
Не найдены совпадающие элементы.
Критическая ошибка — выполнение программы прервано.
```

В этом примере исключение, возникающее при попытке деления на ноль, обрабатывается внутренним блоком try. При этом обеспечивается продолжение выполнения программы. Ошибка, связанная с выходом за границы массива, перехватывается внешним блоком try, который завершает выполнение программы.

Хотя, конечно, использование вложенных операторов try обуславливается не единственной причиной, предыдущая программа создает важный прецедент, который

может быть обобщен. Часто вложенные блоки `try` используются для обработки ошибок разных видов различными способами. Некоторые ошибки являются фатальными и не могут быть исправлены. Другие же являются не столь значительными и могут быть обработаны немедленно. Многие программисты используют внешний блок `try` с целью перехвата большинства серьезных ошибок, позволяя внутреннему блоку `try` обрабатывать менее серьезные. Можно также воспользоваться внешним блоком `try` с оператором `catch` без параметров для перехвата тех ошибок, которые не обрабатываются внутренним блоком.

Минутный практикум

1. Каким образом могут перехватываться все исключения?
2. Может ли один блок `try` использоваться для перехвата двух или большего количества исключений различных типов?
3. Что происходит в случае, если исключение не перехватывается внутренним блоком при использовании вложенных блоков `try`?

Генерирование исключений

В предыдущих примерах производился перехват исключений, автоматически сгенерированных C#. Однако исключение может быть сгенерировано и посредством оператора `throw`. Ниже показан синтаксис этого оператора:

```
throw exceptOb;
```

Здесь `exceptOb` является объектом класса исключений, наследуемым из класса `exception`.

Далее приводится пример, показывающий, каким образом оператор `throw` обеспечивает генерирование исключения `DivideByZeroException`:

```
// Генерирование исключения оператором throw.
using System;

 class ThrowDemo {
  public static void Main() {
     try {
       Console.WriteLine("Перед использованием оператора throw.");
       throw new DivideByZeroException();
     }
     catch (DivideByZeroException) {
       // Перехват исключения.
       Console.WriteLine("Исключение перехвачено.");
     }
     Console.WriteLine("После блока try/catch.");
   }
}
```

Генерирование исключения.

1. Для перехвата всех исключений воспользуйтесь оператором `catch` без указания параметра исключения.

2. Да, воспользуйтесь оператором `catch` для каждого перехватываемого исключения.

3. Исключение, не перехваченное внутренним блоком `try/catch`, перемещается вовне по направлению к внешнему блоку `try`.

Результат выполнения программы:

```
Перед использованием оператора throw.
Исключение перехвачено.
После блока try/catch.
```

Обратите внимание на то, что исключение DivideSyZeroException было создано с помощью модификатора new оператора throw. Помните о том, что throw оперирует с объектами. Поэтому, прежде чем использовать этот оператор, требуется подготовить объект. В рассматриваемом случае для создания объекта DxvideByZeroExceptxon используется конструктор, заданный по умолчанию, однако для создания исключений доступны и другие конструкторы (соответствующий пример приведен далее).

Ответы профессионала

Вопрос. Почему возникает потребность в обеспечении генерирования исключений самим программистом?

Ответ. Такая потребность возникает при использовании исключений из созданных программистом классов исключений. Далее в этой главе будет показано, что создание собственных классов исключений позволяет обрабатывать ошибки в коде в рамках общей стратегии обработки исключений.

Повторное генерирование исключения

Исключение, перехваченное одним оператором `catch`, может генерироваться повторно, благодаря чему оно может перехватываться внешним оператором catch. Наиболее частой причиной повторного генерирования исключений является обеспечение доступа к нему со стороны нескольких обработчиков. Например, один обработчик исключений может обрабатывать один аспект исключения, второй — другой аспект. Для повторного генерирования исключения нужно просто указать ключевое слово `throw` без имени исключения. То есть необходимо воспользоваться следующей формой оператора `throw`:

```
throw;
```

Помните о том, что повторно сгенерированное исключение не может быть повторно перехвачено тем же оператором `catch`. Для повторного перехвата используется следующий оператор `catch`.

Приведенный ниже код иллюстрирует механизм повторного генерирования исключений:

```
// Повторное генерирование исключения.
using System;

class Rethrow {
  public static void genException() {
    // Массив numer содержит больше элементов, чем denom.
    int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i<numer.Lenqth; i++) {
```

```
      try {
        Console.WriteLine(numer[i] + " / " +
                          denom[i] + " будет " +
                          numer[i]/denom[i]);
      }
      catch (DivideByZeroException) {
        // Перехват исключения.
        Console.WriteLine("Нельзя делить на ноль!");
      }
      catch (IndexOutOfRangeException) {
        // Перехват исключения.
        Console.WriteLine("Не найдены совпадающие элементы.");
        throw; // Повторное генерирование исключения.
      }
    }
  }
}

class RethrowDemo {
  public static void Main() {
    try {
      Rethrow.genException();
    }
      catch(IndexOutOfRangeException) {
        // Повторный перехват исключения.
        Console.WriteLine("Критическая ошибка — " +
                          "выполнение программы прекращено.");
    }
  }
}
```

Повторное генерирование исключения.

Перехват повторно сгенерированного исключения.

В данной программе ошибка деления на ноль обрабатывается локально, с помощью метода `genException()`, а ошибка превышения границ массива будет сгенерирована повторно. Исключение перехватывается с помощью метода `Main()`.

Минутный практикум

1. Что выполняет оператор `throw`?
2. По отношению к чему применяется `throw`: к типам или объектам?
3. Может ли исключение генерироваться повторно после того, как оно было перехвачено? Если да, то какая форма оператора `throw` используется в этом случае?

Использование блока `finally`

Иногда требуется определить блок кода, который будет вызываться в то время, когда еще выполняется блок `try/catch`. Исключение может спровоцировать ошибку, прерывающую выполнение текущего метода, вызывая его досрочный возврат. Но этот

1. Оператор `throw` генерирует исключения.

2. Оператор `throw` применяется по отношению к объектам. Эти объекты должны быть экземплярами корректных классов исключений.

3. Да, укажите оператор `throw` без исключения.

метод может открыть файл или сетевое соединение, которые должны быть закрыты. Подобные ситуации довольно часто возникают в программировании, и в C# поддерживается удобный способ их обработки с помощью блока `finally`.

С целью указания блока кода, который вызывается после выхода из блока `try/catch`, поместите блок `finally` в конец последовательности `try/catch`. Общая форма конструкции `try/catch`, включающей блок `finally`, показана ниже:

```
try {
  // Блок кода, выполняющий мониторинг ошибок
}

catch (ExcepType1 exOb1) {
  // Обработка исключения ExcepType1.


catch (ExcepType2 exOb2) {
  // Обработка исключения ExcepType2.
}

finally {
  // Код блока finally.
```

Блок `finally` будет вызываться независимо от того, появится исключение или нет, и независимо от причин возникновения такового. Таким образом, не важно, завершается блок `try` обычным образом либо возникает исключение, — в любом случае выполняется последний блок кода, определенный ключевым словом `finally`.

Ниже приведен пример использования блока `finally`.

```
// Использование блока finally.
using System;

class UseFinally {
  public static void genException(int what) {
    int t;
    int[] nums = new int[2];

    Console.WriteLine("Получение " + what);
    try {
      switch(what) {
        case 0:
          t = 10 / what; // Ошибка деления на ноль.
          break;
        case 1:
          nums[4] = 4; // Ошибка выхода индекса за границы массива.
          break;
        case 2 :
          return; // Возврат из блока try.
      }
    }
    catch (DivideByZeroException) {
      // Перехват исключения.
      Console.WriteLine("Нельзя делить на ноль!");
      return; // Возврат из блока catch.
    }
```

```
    catch (IndexOutOfRangeExcepcion) {
      // Перехват исключения.
      Console.WriteLine("Не найдены совпадающие элементы.");
    }
    finally {
      Console.WriteLine("Выход из блока try.");
    }
  }

class FinallyDemo {
  public static void Main() {

    for(int i=0; i < 3; i++) {
      UseFinally.genException(i);
      Console.WriteLine();
    }
  }
}
```

Вызывается независимо от блоков `try/cdtch`.

Результат выполнения программы:

```
Получение 0
Нельзя делить на ноль!
Выход из блока try.

Получение 1
Не найдены совпадающие элементы.
Выход из блока try.

Получение 2
Выход из блока cry.
```

Как показывает результат выполнения программы, независимо от того, как был осуществлен выход из блока `try`, вызывается блок `finally`.

Более близкое знакомство с исключениями

До настоящего времени производился перехват исключений, но с самими объектами исключений не выполнялись никакие действия. Ранее уже говорилось, что конструкция `catch` позволяет указывать тип исключения и объект исключения. Все исключения наследуются из класса `Exception`.

В классе `Exception` определяется множество свойств. Три наиболее интересных из них называются `Message`, `StackTraee` и `TargecSite`. Все они предназначены только для чтения. Свойство `Message` содержит строку, описывающую природу ошибки. Свойство `SrackTrace` содержит строку, включающую стек вызовов, предшествующих исключению. Свойство `TargetSite` возвращает объект, который указывает метод, генерирующий исключение.

В классе `Exception` также определяется несколько методов. Наиболее часто используемым является метод `ToString()`, который возвращает строку, описывающую исключение. В частности, метод `ToString()` вызывается автоматически в случае, когда исключение отображается с помощью метода `WriteLine()`.

Следующая программа демонстрирует применение описанных свойств и метода:

```
// Использование членов класса Exception.
using System;

class ExcTest {
  public static void genException() {
    int[] nums = new int[4];

    Console.WriteLine("Перед тем как исключение будет сгенерировано.");

    // Генерирование исключения выхода индекса за пределы диапазона.
    nums[7] = 10;
    Console.WriteLine("Этот текст не отображается");
  }
}

class UseExcept {
  public static void Main() {

    try {
      ExcTest.genException();
    }
    catch (IndexOutOfRangeException exc) {
      // catch the exception
      Console.WriteLine("Стандартное сообщение: ");
      Console.WriteLine(exc); // Вызов метода ToString()
      Console.WriteLine("Трассировка стека: " + exc.StackTrace);
      Console.WriteLine("Сообщение: " + exc.Message);
      Console.WriteLine("TargetSite: " + exc.TargetSite);
    }
    Console.WriteLine("После оператора catch.");
  }
}
```

В классе `Exception` определяются четыре конструктора. Два из них будут рассмотрены в дальнейшем:

```
Exception()

Exception(string str)
```

Первый конструктор является конструктором, заданным по умолчанию. Второй указывает свойство `Message`, связанное с исключением. При создании собственных классов исключений требуется реализовывать оба конструктора.

Часто используемые исключения

В пространстве имен `System` определяются многие стандартные встроенные исключения. Все они наследуются из класса `SystemException`, а также генерируются CLR в случае возникновения ошибок в процессе выполнения. Многие из наиболее часто используемых стандартных исключений, определенных в C#, представлены в табл. 10.1.

Таблица 10.1. Наиболее часто используемые исключения, определенные в пространстве имен System

Исключение	Значение
ArrayTypeMismatchException	Тип сохраненного значения несовместим с типом массив
DivideByZeroException	Предпринята попытка деления на ноль.
IndexOutOfRangeException	Индекс массива выходит за пределы диапазона.
invalidCastException	Некорректное преобразование в процессе выполнения.
OutOfMemoryException	Вызов new был неудачным из-за недостатка памяти.
OverflowException	Переполнение при выполнении арифметической операции
StackOverflowException	Переполнение стека.

Минутный практикум

1. Что содержит свойство Message?
2. Когда вызывается код из блока finally?
3. Каким образом отображается трассировка событий, предшествовавших возникновению исключения?

Наследование классов исключений

Несмотря на то что встроенные исключения C# обрабатывают большинство распространенных ошибок, механизм обработки исключений C# не ограничивается Этими ошибками. На самом деле мощь подхода C# к обработке исключений проявляется в способности к обработке исключений, заданных программистом. Можно создавать заказные исключения, выполняющие обработку ошибок в пользовательском коде. Генерирование исключений не представляет особых сложностей. Просто определите класс, наследуемый из класса Exception. В качестве общего правила руководствуйтесь тем, что определенные пользователем исключения наследуются из класса ApplicationException, так как они представляют собой иерархию зарезервированных исключений, связанных с приложениями. Наследуемые классы не нуждаются в фактической реализации в каком-либо виде, поскольку само их существование в системе типов данных позволяет воспользоваться ими в качестве исключений.

Создаваемые пользователем классы исключений автоматически получают доступные для них свойства и методы, определенные в классе Exception. Конечно, можно переопределять один или больше членов в создаваемых вами классах исключений.

Далее будет рассмотрен пример, в котором создается исключение NonIntResultException, определяющее два стандартных конструктора, а также переопределяющее метод ToString().

```
// Использование заказного исключения.
using System;
```

1. Свойство Message содержит текст, описывающий исключение.

2. Блок finally является последней конструкцией, вызываемой после выхода из блока try.

3. Для вывода на экран трассировки стека отобразите значение свойства StackTrace, определяемого в классе Exception.

```
// Создание исключения.
class NonIntResultException : ApplicationException {
  // Реализация стандартных конструкторов.
  public NonIntResultException() : base() { }
  public NonIntResultException(string str) : base(str) { }

  // Переопределение ToString для NonIntResultException.
  public override string ToString() {
    return Message;
  }
}

class CustomExceptDemo {
  public static void Main() {

      // Здесь numer содержит некоторые нечетные значения.
         int[] numer = { 4, 8, 15, 32, 64, 127, 256, 512 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i<numer.Length; i++) {
      try {
        if ((numer[i]%2) != 0)
          throw new
            NonIntResultException("Результат " +
                  numer[i]+ "/" + denom [i] + " нечетный.");

        Console.WriteLine(numer[i] + " / " +
                          denom[i] + " будет " +
                          numer[i]/denom[i]);
      }
      catch (DivideByZeroException) {
        // Перехват исключения.
        Console.WriteLine("Нельзя делить на ноль!");
      }
      catch (IndexOutOfRangeException) {
        // Перехват исключения.
        Console.WriteLine("Не найдены совпадающие элементы.");
      }
      catch (NonIntResultException exc) {
        Console.WriteLine(exc);
      }
    }
  }
}
```

Заказное исключение.

Переопределение метода `ToString()`.

Генерирование заказного исключения.

Результат выполнения программы:

```
4/2 будет 2
Нельзя делить на ноль!
Результат 15/4 нечетный.
32 / 4 будет 8
Нельзя делить на ноль!
Результат 127 / 8 нечетный.
Не найдены совпадающие элементы.
Не найдены совпадающие элементы.
```

Перед тем как перейти к дальнейшему изложению, немного поэкспериментируем с программой. Например, попытайтесь превратить в комментарий переопределение метода `ToString()` и понаблюдайте за результатами. Либо попытайтесь создать исключение, воспользовавшись заданным по умолчанию конструктором, а затем посмотрите, как C# генерирует это исключение в качестве сообщения, заданного по умолчанию.

Перехват исключений производного класса

При попытках перехвата типов исключений, относящихся к наследуемому и наследующему классу, необходимо проявлять осторожность в случае использования операторов `catch`. Это связано с тем, что конструкция `catch` для наследуемого класса совпадает с такой же конструкцией для любого наследующего класса. К примеру поскольку наследуемым классом для всех исключений является класс `Exception`, следовательно, конструкции `catch` из класса `Exception` будут перехватывать все возможные исключения. Конечно, использование конструкции `catch` без аргументов обеспечивает более «прозрачный» способ для перехвата всех исключений, чем описывалось ранее. Однако проблема перехвата исключений наследующих классов является весьма важной в другом контексте, особенно когда пользователь создает свои собственные исключения.

Если требуется перехватывать исключения в наследуемом и наследующем классах одновременно, в последовательности `catch` наследующий класс будет первым. Если же этого не сделать, конструкция `catch` из наследуемого класса будет также выполнять перехват во всех наследующих классах. Этого правила следует придерживаться, ибо помещение наследуемого класса первым приведет к созданию недостижимого кода, когда конструкция `catch` из наследующего класса никогда не будет вызываться. В C# недостижимая конструкция `catch` приведет к появлению ошибки.

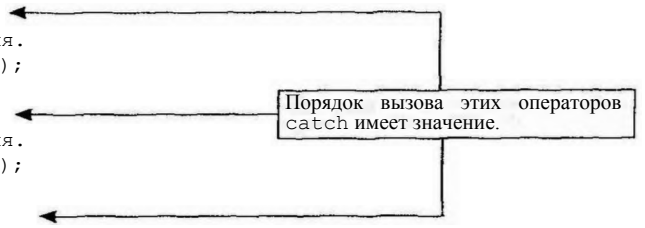
В следующей программе создаются два класса исключений, `ExceptA` и `ExceptB`. Класс `ExceptA` наследуется из класса `ApplicationException`. Класс `ExceptB` наследуется из класса `ExceptA`. Затем эта программа генерирует исключения обоих типов.

```
// Исключения наследующего класса должны следовать перед исключениями
// наследуемого класса.
using System;

// Создание исключения.
class ExceptA : ApplicationException [
  public ExceptA() : base() { }
  public ExceptA(string str) : base(str) { }

  public override string ToString() {
     return Message;
   }


// Создание исключения, наследующего исключение ExceptA
class ExceptB : ExceptA {
  public ExceptBO ; base() { }
  public ExceptB(string str) : base(str) { }
```

Наследование исключения.

Наследование исключения из `ExceptA`.

```
  public override string ToString() {
    return Message;
  }
}

class OrderMatters {
  public static void Main() {
    for(int x = 0; x < 3; x++) {
      try {
        if(x==0) throw new ExceptA("Перехвачено исключение ExceptA");
        else if(x==1) throw new ExceptB("Перехвачено исключение ExceptB");
        else throw new Exception();
      }
      catch (ExceptB exc) {
        // Перехват исключения.
        Console.WriteLine(exc);
      }
      catch (ExceptA exc) {
        // Перехват исключения.
        Console.WriteLine(exc);
      }
      catch (Exception exc) {
        Console.WriteLine(exc);
      }
    }
  }
}
```



Ответы профессионала

Вопрос. Поскольку исключения обычно проявляются в виде какой-либо ошибки, почему так важен перехват исключений наследуемого класса?

Ответ. Конструкция `catch`, перехватывающая исключения наследуемого класса, позволяет организовать перехват целой категории исключений, обеспечить их обработку с помощью одного оператора `catch`, а также избежать дублирования кода. Например, можно создать набор исключений, описывающих аппаратные ошибки некоторого рода. Если обработчик исключений просто сообщает пользователю о том, что произошли аппаратные ошибки, можно воспользоваться общей конструкцией `catch` с целью перехвата всех исключений данного типа. Обработчик может просто отображать строку `Message`. Поскольку используемый в данном случае код будет одинаковым для всех исключений, одна конструкция `catch` может обрабатывать все аппаратные исключения.

Результат выполнения программы будет таким:

```
Перехвачено исключение ExceptA
Перехвачено исключение ExceptB
System.Exception: Exception of type System.Exception was thrown
  at OrderMatters.Main()
```

Обратите внимание на порядок следования операторов `catch`. Этот порядок является существенным. Поскольку исключение `ExceptB` наследуется из исключения `ExceptA`, оператор `catch` для исключения `ExceptB` должен следовать прежде оператора

catch для исключения ExceptA. Аналогично, оператор catch для класса Exception (который является наследуемым классом для всех исключений) должен отображаться последним. Если вы хотите проверить это утверждение на практике, попытайтесь переставить местами операторы catch. В результате произойдет ошибка компиляции.

Проект 10-1. Добавление исключений в класс Queue

QExcDemo.cs

В ходе осуществления этого проекта будут созданы два класса исключений, которые могут использоваться классами очередей, разработанными в проекте 9-1. Эти классы отображают ошибки состояний, свидетельствующие о том, что очередь пуста или заполнена. Данные исключения вызываются методами put() и get() в случае возникновения соответствующих ошибок. С целью упрощения эти исключения в разрабатываемом проекте включены лишь в один класс FixedQueue, но пользователь может легко включить исключения в другие классы очередей из проекта 9-1.

Пошаговая инструкция

1. Создайте файл QExcDemo.cs.
2. В файле QExcDemo.cs определите следующие исключения:

```
/*
    Проект 10-1

    Добавление обработки исключений в классы очередей.
*/
using System;

// Исключение для ошибок заполнения очереди.
class QueueFullException : ApplicationException {
  public QueueFullException() : based { }
  public QueueFullException(string str) : base(str) { }

  public override string ToString() {
    return "\n" + Message;
   }
}

// Исключение для ошибок пустой очереди.
class QueueEmptyException : ApplicationException {
  public QueueEmptyException() : based { }
  public QueueEmptyException(string str) : base(str) { }

  public override string ToString() {
    return "\n" Message;
   }
}
```

Исключение QueueFullException генерируется в случае, если предпринимается попытка сохранения элемента в уже заполненной очереди. Исключение QueueEmptyException генерируется в случае, если предпринимается попытка удаления элемента из пустой очереди.

3. Измените класс FixedQueue таким образом, чтобы он генерировал исключение в случае возникновения ошибки, как показано ниже. Добавьте его в класс QExcDemo.cs.

```
// Класс символов фиксированного размера, использующий исключения.
class FixedQueue : ICharQ {
  char[ ] q;  // Этот массив содержит очередь.
  int putloc, getloc; // Индикаторы put и get

  // Конструирование пустой очереди заданного размера.
  public FixcaQueue(int size) {
    q = new char(size + l]; // Выделение памяти для очереди.
    putloc = getloc = 0;
  }

  // Помещение символа в очередь.
  public void put(char ch) {
    if (putloc==q.Length-1)
      throw new QueueFullException("Максимальная длина " +
                                          (q.Length-1));

    putloc++;
    q[putloc] = ch;
  }

  // Получение символа из очереди.
  public char get() {
    if(getloc == putloc)
      throw new QueueEmptyException();

    getloc++;
    return q[getloc];
  }
}
```

Благодаря добавлению исключений в класс FixedQueue можно обрабатывать ошибки очереди наиболее рациональным способом. Как вы помните, предыдущая версия FixedQueue просто отображала сообщение об ошибке. Генерирование исключений более эффективно, поскольку в этом случае код, использующий класс FixedQueue, может еще и обрабатывать ошибки.

4. Для проверки возможностей обновленного класса FixedQueue просто добавьте указанный ниже код класса QExcDemo в файл QExcDemo.cs.

```
// Демонстрация возможностей исключений, относящихся к очереди.

class QExcDemo {
  public static void Main() {
    FixedQueue q = new FixedQueue(10);
    char ch;
    int i;

    try {
      // Переполнение очереди.
      for(i=0; i < 11; i++) {
        Console.Write("Попытка сохранения: " +
                          (char) ( 'A' + i));
        q.put((char) ('A' + i));
        Console.WriteLine(" = OK");
```

```
        }
        Console.WriteLine();
      }
      catch (QueueFullException exc) {
        Console.WriteLine(exc);
      }
      Console.WriteLine();

      try {
        // Попытка чтения из пустой очереди.
        for(i=0; i < 11; i++) {
          Console.Write("Получение следующего символа: ");
          ch = q.get();
          Console.WriteLine(ch);
        }
      }
      catch (QueueEmptyException exc) {
        Console.WriteLine(exc);
      }
    }
  }
```

5. Для создания программы потребуется скомпилировать файл `QExcDemo.cs` совместно с файлом `IQChar.cs`. Файл `IQChar.cs` содержит код интерфейса очереди. При запуске на выполнение программы `QExcDemo` получается следующий результат:

```
Попытка сохранения: A - OK
Попытка сохранения: B - OK
Попытка сохранения: C — OK
Попытка сохранения: D - OK
Попытка сохранения: E - OK
Попытка сохранения: F — OK
Попытка сохранения: G - OK
Попытка сохранения: H - OK
Попытка сохранения: I - OK
Попытка сохранения: J - OK
Попытка сохранения: K
Максимальная длина 10

Получение следующего символа: A
Получение следующего символа: B
Получение следующего символа: C
Получение следующего символа: D
Получение следующего символа: E
Получение следующего символа: F
Получение следующего символа: G
Получение следующего символа: H
Получение следующего символа: I
Получение следующего символа: J
Получение следующего символа:
An exception of type QueueEmptyException was thrown
```

Использование ключевых слов checked и unchecked

В ходе выполнения арифметических вычислений могут возникать ошибки переполнения. Обратите внимание на представленный ниже пример.

```
byte a, b, result;
a = 127;
b = 127;

result = (byte) (a * b)
```

В этом примере произведение .множителей a и b превышает допустимый диапазон знамения byte. В результате произошла ошибка переполнения.

В C# можно задавать вызов исключений в случае переполнения, воспользовавшись ключевыми словами checked и unchecked. Для указания выражения, проверяемого и случае переполнения, используется ключевое слово checked. Для игнорирования переполнения используется ключевое слово unchecked. В последнем случае происходит усечение результата с тем, чтобы тип результата совпадал с типом целевого выражения.

Оператор checked имеет две основных формы. Одна форма используется для проверки указанного выражения. Вторая форма используется для проверки блока операторов.

```
checked(expr)

checked {
   // Проверяемые операторы.
}
```

В этом фрагменте кода параметр expr определяет проверяемое выражение. Если при вычислении выражения возникает переполнение, генерируется исключение OverflowException.

Оператор unchecked имеет три основных формы. При использовании первой формы игнорируется переполнение для указанного исключения. Вторая форма игнорирует переполнение для блока операторов.

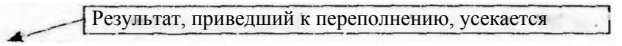
```
unchecked (expr)

unchecked {
   // Операторы, для которых будет проигнорировано переполнение.
}
```

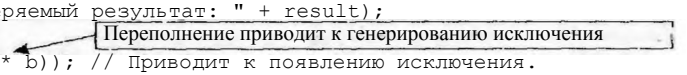
В этом фрагменте кода параметр expr определяет выражение, которое не будет проверяться на предмет возникновения ситуации переполнения. Если при вычислении подобного выражения возникает ситуация переполнения, происходит усечение результата.

Приведенная ниже программа демонстрирует механизм использования ключевых слои checked и unchecked.

```
// Механизм использования ключевых слов checked и unchecked.
using System;
class CheckedDeno {
  public static void Main() {
    byte a, b;
    byte result;

    a = 127;
    b = 127;
```

```
      try {
       result - unchecked;
       Console.WriteLine("Непроверяемый результат: " + result);
       result = checked((byte)(a * b)); // Приводит к появлению исключения.
       Console.WriteLine("Проверяемый результат: " + result); // Не может
                                                              // выполняться

      catch (OverflowException exc) {
        // Перехват исключения
       Console.WriteLine(exc);
     }
   }
}
```





Результат выполнения программы:

```
Непроверяемый результат: 1
System.OverflowException: Exception of type
                          System.OverflowException was thrown.
                          at CheckeaDemo.Main()
```

В ходе выполнения программы результат вычисления непроверяемою выражения усекается. Переполнение, вызываемое проверяемым выражением, приводит к генерированию исключения.

В предыдущей программе показано использование ключевых слов `checked` и `unchecked` в составе единственного выражения. В следующей программе демонстрируется проверка и отсутствие проверки для блока операторов:

```
// Использование ключевых слов checked и unchecked при работе с блоками
    // операторов.
using System;

class CheckedBlooks {
    public static void Main();
    int a, b;
    type result;

     a = 127;
       b = 127;


       unchecked {
         a = 127;
         b = 127.
         result = uncheked((byte)(a * b));
         Console.WriteLine("Непроверяемый результат: " + result);

         a = 128;
         b = 6;
         result = unchecked((byte)(a * b));
         Console.WriteLine("Непроверяемый результат: " + result);
       }

       checked {
         a = 2 ;
```

```
        b = 7;
        result = checked((byte)(a * b)); // Все в порядке.
        Console.WriteLine("Проверяемый результат: " + result);

        a = 127;
        b = 127;
        result = checked((byte)(a * b)); // Генерируется исключение.
        Console.WriteLine("Проверяемый результат: " - result); // Не может
                                                               // выполняться.
      }

    catch (OvertflowException exc) {
      // Перехват исключения.
      Console.WriteLine(exc);
    }
  }
}
```

Результат выполнения программы:

```
Непроверяемый результат: 1
Непроверяемый результат: 113
Проверяемый результат: 14
Sustem.OverflowException: Exception of type
                Systen.OverflowException was thrown.
                at ChackedBlocks.Main()
```

Как видите, в случае возникновения переполнения в непроверяемых блоках производится усечение результата. Если же переполнение происходит в проверяемом блоке, генерируется исключение.

Единственная причина, в силу которой требуется использовать ключевые слова `checked` и `unchecked`, заключается в том, что статус переполнения проверяемый/непроверяемый определяется путем установки соответствующей опции компилятора, а также с помощью самой среды выполнения. Поэтому в некоторых программах лучше явно указать статус проверки переполнения.

Ответы профессионала

Вопрос. Когда следует использовать обработку исключений в программах? Когда нужно создавать пользовательские, заказные классы исключений?

Ответ. Поскольку в C# повсеместно применяются исключения с целью сообщения о возникающих ошибках, практически все прикладные программы используют механизм обработки исключений. Обработка исключений, основанная на использовании встроенных исключений C#, является наиболее простой для освоения начинающими программистами. Труднее принять решение относительно того, когда и как следует использовать пользовательские, настраиваемые исключения. Существуют два общих способа создания отчетов о возникающих ошибках: возвращаемые значения и исключения. Какой же подход лучше? Несомненно, при программировании на языке C# использование механизма обработки исключений должно быть нормой. Хотя возврат кода ошибки и может стать альтернативой в некоторых случаях, исключения обеспечивают более совершенный и более наглядный способ обработки ошибок. Именно этот способ должны выбирать профессиональные программисты на C#, выполняющие обработку ошибок в своих программах.

Контрольные вопросы

1. Какой класс находится на вершине иерархии исключений?
2. Кратко опишите методику использования операторов `try` и `catch`.
3. Найдите ошибку в следующем фрагменте кода:

```
// ...
vals[18] = 10;
catch (IndexOutOfRangeException exc) {
  // Обработка ошибки.
}
```

4. Что произойдет, если исключение не будет перехвачено?
5. Найдите ошибку в следующем фрагменте кода:

```
class A : Exception { ...

class Б : A { ...

// ...

try {
  // ...
}
catch (A exc) { ... }
catch (B exc) { ... }
```

6. Может ли исключение, перехваченное внутренним блоком `catch`, повторно генерироваться в качестве исключения для внешнего блока `catch`?
7. Является ли блок `finally` последним фрагментом кода, выполняемым перед завершением программы? Обоснуйте свой ответ.
8. В упражнении 3 раздела «Контрольные вопросы» главы 6 был создан класс `Stack`. Добавьте в этот класс заказные исключения, которые сообщают о состояниях пустого и заполненного стека.
9. Расскажите о назначении ключевых слов `checked` и `unchecked`.
10. Каким образом могут быть перехвачены все исключения?

Ввод/вывод

- Поток
- Классы потоков
- Консольный ввод/вывод
- Ввод/вывод в файлы
- Чтение и запись двоичных данных
- Произвольный доступ к содержимому файла
- Преобразование числовых строк

С самого начала этой книги в программах используется подсистема ввода/вывода C# Console.WriteLine. Однако пока она не была четко описана. Поскольку при вводе/выводе в используется иерархия классов, невозможно представить теорию в детали этой подсистемы без предварительного описания классов. механизма наследования и исключений. Теперь мы готовы к изучению этой темы. Как отмечалось в главе 1, в C# используются подсистема в вода/вывода и классы, определенные в среде .NET Framework. Таким образом, описывая подсистему ввода/вывода в C#, мы фактически описываем подсистему ввода/вывода платформы .NET.

В главе рассматриваются механизмы консольного ввода/вывода и ввода/вывода в файлы. Следует учесть, что подсистема ввода/вывода в C# является весьма развитой. В главе рассматриваются наиболее важные и широко используемые возможности этой подсистемы, однако некоторые аспекты использования механизма ввода/вывода придемся изучать самостоятельно. К счастью, подсистема ввода/вывода C# довольно логична. Как только вы поймете основы этой подсистемы, освоить детали станет значительно проще.

Потоки ввода/вывода C#

Программы C# реализуют ввод/вывод с помощью потоков. Под *потоком* понимается вывод либо получение информации. Поток связан с физическим устройством посредством системы ввода/вывода C#. Все потоки ведут себя одинаково, в то время как реальные физические устройства, которые с ними связаны, существенно отличаются друг от друга. Поэтому классы и методы ввода/вывода могут применяться совместно с устройствами многих типов. Например, методы, используемые для вывода на консоль, могут также использоваться для записи данных в файл на диске.

Байтовые потоки и потоки символов

На самом низком уровне все операции ввода/вывода в C# оперируют байтами. Подобный подход имеет смысл, поскольку большинство устройств, предназначенных для выполнения операций ввода/вывода, являются байт-ориентированными. Однако данным при общении с компьютером удобнее использовать символы. Напомним, что в C# тип данных char имеет разрядность 16 битов, а тип данных byte является 8-битовым. Если применяется набор символов ASCII, довольно просто выполнить преобразование между типами char и byte при этом для величины типа char нужно игнорировать старший байт. Данный прием не подходит при работе с остальными символами Unicode, которые используют оба байта. Поэтому байтовые потоки не вполне удобно использовать при выполнении символьного ввода, вывода. Для решения этой задачи в C# определены несколько классов, преобразующих байтовый поток в поток символов, то есть автоматически преобразующих данные типа byte в данные типа char и наоборот.

Предопределенные потоки

(начало не разобрать)

по умолчанию это консоль. Так, при вызове метода `Console.WriteLine()` информация автоматически направляется в поток `Console.Out`. Поток `Console.In` связан со стандартным устройством ввода, которым по умолчанию является клавиатура. Однако эти потоки могут быть перенаправлены на любое соответствующее устройство ввода/вывода. Стандартные потоки являются символьными потоками. Поэтому они тывают и записывают символы.

Минутный практикум

1. Что в C# понимается под потоком?
2. Назовите типы потоков C#.
3. Какие потоки являются предопределенными?

Классы потоков

В C# определяются классы байтовых потоков и классы потоков символов. Однако классы потоков символов в действительности представляют собой оболочку, которая преобразует передаваемый на более низком уровне поток байтов в поток символов. Причем выполнение этого преобразования происходит автоматически. Таким образом, символьные потоки построены на основе байтовых потоков и являются их логическим дополнением.

Те классы потоков определены в пространстве имен `System.IO`. Для применения этих классов в начале программы обычно помещают следующий оператор:

```
ing System.IO;
```

Нет необходимости в указании класса `System.IO` для консольного ввода/вывода. Класс `Console` определен в пространстве имен `System`.

Класс `Stream`

Потоки в C# создаются с помощью класса `System.IO.Stream`. Класс `Stream` представляет байтовые потоки и является базовым классом для всех других классов потоков. Он также является абстрактным, поэтому не существует реализации объекта `Stream`. Класс `Stream` определяет набор стандартных операций с потоками. В табл. 11.1 описано несколько распространенных методов, определенных в классе `Stream`.

Таблица 11.1. Некоторые методы, определенные в классе `Stream`

Метод	Описание
`void Close()`	Закрывает поток
`void Flush()`	Записывает содержимое потока в физическое устройство
`int ReadByte()`	Возвращает целочисленное представление следующего доступного байта в потоке ввода. По достижении конца файла возвращает -1

1. Под потоком в C# понимается вывод либо получение данных.
2. В C# есть два типа потоков — байтовые и потоки символов.
3. Предопределенными потоками являются потоки `Console.In`, `Console.Out` и `Console.Error`.

Метод	Описание
`int Read(byte[] buf, int offset, int numBytes)`	Осуществляет попытку чтения *numBytes* байтов из потока ввода и помещает их в массив *buf*, начиная с элемента *buf[offset]*. При этом возвращается количество успешно считанных байтов
`long Seek (long offset, SeekOrigin origin)`	Задает позицию текущего байта в потоке, используя значение смещения *offset* относительно позиции *origin*
`void WriteByte(byte b)`	Осуществляет запись одного байта в поток вывода
`int Write(byte[] buf, int offset, int numBytes)`	Осуществляет запись *numBytes* байтов в поток вывода из массива *buf*, начиная с элемента *buf[offset]*. При этом возвращается количество записанных байтов

Как правило, если появляется ошибка ввода/вывода, методы, приведенные в табл. 11.1, генерируют исключение `IOException`. Если предпринимается попытка выполнения недопустимой операции (например, попытка записи информации в поток, предназначенный только для чтения), генерируется исключение `NotSupportedException`.

Обратите внимание на то, что в классе `Stream` определены методы, предназначенные для записи и чтения данных. Однако не все потоки поддерживают и запись, и чтение, поскольку можно открывать потоки, предназначенные только для чтения или только для записи. Также не все потоки поддерживают позиционирование методом `Seek()`. С целью определения возможностей конкретного потока используется одно или несколько свойств класса `Stream`. Эти свойства приведены в табл. 11.2.

Таблица 11.2. Свойства, определенные в классе `Stream`

Свойство	Описание
`bool CanRead`	Свойство имеет значение `true`, если возможно чтение из потока. Свойство предназначено только для чтения
`bool CanSeek`	Свойство имеет значение `true`, если поток позволяет задавать текущую позицию элемента. Свойство предназначено только для чтения
`bool CanWrite`	Свойство имеет значение `true`, если возможна запись в поток. Свойство предназначено только для чтения
`long Length`	Свойство указывает длину потока. Свойство предназначено только для чтения
`long Position`	Свойство представляет позицию текущего элемента потока. Свойство предназначено как для чтения, так и для записи

Классы байтовых потоков

Из класса `Stream` наследуются три класса байтовых потоков, перечисленные ниже.

Класс	Описание
`BufferedStream`	Обеспечивает буферизацию байтового потока. В большинстве случаев буферизация позволяет увеличить производительность
`FuleStream`	Байтовый поток, предназначенный для осуществления операций ввода/вывода в файл
`MemoryStream`	Байтовый поток, использующий память в качестве хранилища

Программист может определять и собственные классы потоков. Однако для подавляющего большинства приложений вполне достаточно встроенных потоков.

Классы потоков символов

В верхней части иерархии символьных потоков находятся абстрактные классы `TextReader` и `TextWriter`. Методы, определенные в этих классах, обеспечивают минимальный набор функций ввода/вывода для всех символьных потоков.

В табл. 11.3 приведены методы ввода класса `TextReader`. Обычно в случае возникновения ошибки эти методы генерируют исключение `IOException`. (Иногда возможно возникновение других исключений.) Особый интерес представляет метод `ReadLine()`, который полностью считывает строку текста, возвратам ее в виде строки C# (данных типа `string`). Этот метод удобно использовать при чтении данных из потока ввода, содержащих пробелы.

Таблица 11.3. Методы ввода, определенные в классе `TextReader`

Метод	Описание
`void Close()`	Закрывает поток ввода
`int Peek()`	Получает следующий символ из потока ввода, но не удаляет его. Возвращает - 1, если нет доступного символа
`int Read()`	Возвращает целочисленное представление следующего доступного символа из потока ввода. Возвращает -1 в случае достижения конца потока
`int Read(char[] buf, int offset, int numChars)`	Осуществляет попытку чтения *numChars* символов из потока ввода и помещает их в массив *buf,* начиная с элемента *buf[offset].* При этом возвращается количество успешно считанных символов
`int ReadBiock (char[] buf, int offset, int numChars)`	Осуществляет попытку чтения *numChars* символов из потока ввода и помещает их в массив *buf,* начиная с элемента *buf[offset].* При этом возвращается количество успешно считанных символов
`string ReadLine()`	Считывает строку текста и возвращает ее в виде строки C#. Нулевое значение возвращается в том случае, если был считан символ конца файла
`string ReadToEnd()`	Считывает все оставшиеся символы потока и возвращает их в виде строки C#

В классе `TextWriter` определены различные версии методов `Write()` и `WriteLine()`, которые позволяют выводить данные встроенных типов. Ниже приводится лишь несколько перегруженных версий этих методов.

Метод	Описание
`void Writ(int val)`	Запись значения типа `int`
`void Write(double val)`	Запись значения типа `double`
`void Write(bool val)`	Запись значения типа `bool`
`void WriteLine(string val)`	Запись строки, завершаемой символами перехода на новую строку
`void WriteLine(uint val)`	Запись значения типа `uint` и символов перехода на новую строку
`void Wr1toLine(char vat)`	Запись символа, за которым следуют символы перехода на новую строку

В дополнение к методам `Write()` и `WriteLine()` в классе `TextWriter` также определены методы `Close()` и `Flush()`:

```
virtual void Close()
virtual void Flush()
```

Метод `Flush()` используется для вывода на физический носитель данных, которые остались в буфере вывода. Метод `Close()` закрывает поток.

Классы `TextReader` и `TextWriter` реализуются с помощью перечисленных ниже классов символьных потоков. Поэтому данные потоки поддерживают методы и свойства, определенные в классах `TextReader` и `TextWriter`.

Класс потока	Описание
`StreamReader`	Считывает символы из байтового потока. Этот класс является оболочкой байтового потока ввода
`StreamWriter`	Записывает символы в байтовый поток. Этот класс является оболочкой байтового потока вывода
`StringReader`	Считывает символы из строки
`StringWriter`	Записывает символы в строку

Двоичные потоки

В дополнение к байтовым и символьным потокам в C# определяются два двоичных класса потоков, которые могут использоваться для непосредственного считывания и записи двоичных данных. Эти потоки называются `BinaryReader` и `BinaryWriter`. Они подробно рассматриваются далее в этой главе при описании механизма ввода/вывода в двоичные файлы.

Теперь, когда вы получили общее представление о структуре подсистемы ввода/вывода в C#, приступим к детальному изучению компонентов этой структуры. Сначала рассмотрим подсистему консольного ввода/вывода.

Минутный практикум

1. Какой класс находится в верхней части иерархии потоков?
2. Назовите свойства класса `Stream`.
3. Какие классы находятся вверху иерархии классов символьных потоков?

Консольный ввод/вывод

Консольный ввод/вывод реализуется с помощью стандартных потоков `Console.In`, `Console.Out` и `Console.Error`. Консольный ввод/вывод рассматривался в главе 1. поэтому вы с ним уже знакомы. Как будет показано далее, этот ввод/вывод обладает некоторыми дополнительными возможностями.

---

1. В верхней части иерархии потоков находится класс `Stream`.

2. Это следующие свойства: `CanSeek`, `CanRead`, `CanWrite`, `Length` и `Position`.

3. Вверху иерархии классов символьных потоков находятся классы `TextReader` и `TextWriter`.

Однако, прежде чем приступить к изложению дальнейшего материала, следует еще раз подчеркнуть следующий момент: большинство реальных приложений C# не являются консольными приложениями, ориентированными на обработку текста. В данном случае речь идет скорее о графических программах либо компонентах, использующих оконный интерфейс для организации взаимодействия с пользователем. Поэтому в C#-программах не столь широко применяется консольный ввод/вывод. Хотя программы, имеющие интерфейс в виде текстовой строки, представляют собой прекрасные учебные примеры и могут пригодиться в качестве компактных утилит, они не подходят в качестве реальных коммерческих программ.

Чтение данных с консоли

Поток `Console.In` является экземпляром класса `TextReader` и для получения доступа к нему предназначены методы и свойства, определенные в классе `TextReader`. Однако обычно используются методы класса `Console`, с помощью которых выполняется автоматическое считывание данных из потока `Console.In`. В классе `Console` определяются два метода ввода: `Read()` и `ReadLine()`.

Для чтения одного символа применяется метод `Read()`.

```
static int Read()
```

Этот метод мы использовали в главе 3. Метод возвращает очередной символ, считанный с консоли. Возвращаемый символ имеет тип `int`, и требуется его преобразование в тип `char`. При возникновении ошибки возвращается значение -1. Если возникает сбой, метод генерирует исключение `IOException`.

Для считывания строки символов используется метод `ReadLine()`:

```
static string ReadLine()
```

Этот метод считывает введенные символы до тех пор, пока не будет нажата клавиша [Enter]. Считанные символы возвращаются в виде объекта типа `string`. При сбое также генерируется исключение `IOException`.

Ниже приводится программа, демонстрирующая процесс чтения массива символов из потока `Console.In`.

```
// Ввод с консоли посредством метода ReadLine().
using System;

class ReadChars {
  public static void Main() {
    string str;

    Console.WriteLine("Введите несколько символов.");
    str = Console.ReadLxne();
    Console.WriteLine("Вы ввели: " + str);
  }
}
```

Чтение строки, введенной с клавиатуры.

Результат выполнения программы:

```
Введите несколько символов.
Это тест.
Вы ввели: Это тест.
```

Несмотря на то что методы класса Console реализуют наиболее простой способ чтения потока Console.In, можно вызывать методы, находящиеся на более низком уровне — в классе TextReader. Ниже приведен текст предыдущей программы, переписанной с использованием методов, определенных в классе TextReader.

```
/* Непосредственное использование потока Console.In для чтения массива
   байтов с клавиатуры. */
using System;

      class ReadChars2 {
  public static void Main() {
    string str;

    Console.WriteLine("Введите несколько символов.");

    str = Console.In.ReadLine();

    Console.WriteLine("Вы ввели: " + str);
  }
}
```

Непосредственное чтение из потока Console.In

Обратите внимание на то, как вызывается метод ReadLine() для потока Console.In.

Вывод данных на консоль

Потоки Console.Out и Console.Error являются объектами типа TextWriter. Вывод данных на консоль проще всего реализовать посредством уже знакомых вам методов Write() и WriteLine(). Различные версии этих методов предназначены для вывода данных каждого из встроенных типов. В классе Console определены собственные версии методов Write() и WriteLine(), поэтому их можно вызвать непосредственно через этот класс. Однако эти (и другие) методы можно вызвать и через класс TextWriter, который находится на более низком уровне.

Ниже приводится программа, демонстрирующая выполнение записи в потоки Console.Out и Console.Error.

```
   // Запись данных в потоки Console.Out и Console.Error.
using System;

 class ErrOut {
  public static void Main() {
    int a=10, b=0;
    int result;

    Console.Out.WriteLine("Генерирование исключения.");
    try {
      result = a / b; // Генерирование исключения.
     } catch (DivideByZeroException exc) {
       Console.Error.WriteLine(exc.Message);
     }
   }
 }
```

Запись в потоки Console.Out и Console.Error.

Результат выполнения программы:

```
Генерирование исключения.
Попытка деления на нуль.
```

Иногда пользователи. имеющие небольшой опыт в программировании, не могут решить, когда следует применять поток `Console.Error`. Поскольку потоки `Console.Out` и `Console.Error` по умолчанию осуществляют вывод на консоль, зачем нужны два различных потока? При ответе на этот вопрос следует учесть тот факт, что стандартные потоки можно перенаправить на другие не устройства ввода/вывода. Например, содержимое потока `Console.Error` можно направить вместо экрана в файл на диске. Тогда сообщения об ошибках вывода, например, будут написаны в журнальный файл, а обычные сообщения программы отобразятся на экране. Кстати, если вывод на консоль перенаправлен, а поток ошибок нет, то сообщения об ошибках можно увидеть на консоли во время работы программы. Процедура перенаправления потоков будет рассмотрена далее, после описания операций ввода/вывода в файлы.

Класс `FileStream` и байт-ориентированный ввод/вывод в файлы

В C# поддерживаются классы, обеспечивающие операции чтения/записи для файлов. Конечно, наиболее распространенным типом файла является дисковый файл. На уровне операционной системы все файлы рассматриваются как двоичные файлы. Как и следовало ожидать, в C# поддерживанием методы чтения и записи байтов при работе с файлами. Поэтому достаточно широко распространены операции чтения/записи файлов, при осуществлении которых используются потоки батов. В C# обеспечивается заключение файлового потока, ориентированного на использование байтов в символьный поток. Операции с символьными файлами удобны, если требуется сохранять текст. Символьные потоки будут описаны далее в этой главе, Сейчас же мы расскажем о механизме ввода/вывода, ориентированном на использование байтов.

Для создания байтового потока, связанного с файлом, используется класс `FileStream`. Этот класс наследуется из класса `Stream` и поэтому имеет все свойства, присущие классу `Stream`.

Помните, что классы потоков (в том числе и класс `FileStream`) определены в `System.IO`. Поэтому обычно в начало любой программы, применяющей классы потоков, включается конструкция.

```
using System.IO;
```

Открытие и закрытие файла

При создании байтовою потока, связанного с файлом, требуется сформировать объект `FileStream`. Из этою объекта наследуется несколько конструкторов. Наиболее часто используется следующий конструктор:

```
FileStream(string filename, FileMode)
```

где `filename` указывает имя открываемого файла, которое может включать полностью определенный путь к нему. Параметр `mode` определяет, каким образом открывается

файл. Здесь необходимо указать одно из значений, определенных в перечислении FileMode. Эти значения описаны в табл. 11.4.

Если попытка открытия файла была неудачной, генерируется исключение. Если файл не может быть открыт, поскольку он не существует, генерируется исключение FileNotFoundException.

Таблица 11.4. Значения перечисления FileMode

Значение	Описание
FileMode.Append	Выводимые данные добавляются после данных, находящихся в файле
FileMode.Create	Создается новый выходной файл. Любой ранее созданный файл с аналогичным именем уничтожается
FileMode.CreateNew	Создается новый выходной файл
FileMode.Open	Открывается существующий файл
File.Mode.OpenOrCreate	В случае наличия файла происходит его открытие, если файл не существует, он создается
FileMode.Truncate	Открывается существующий файл, но его длина усекается до нуля

Если из-за ошибки ввода/вывода файл не открывается, генерируется исключение IOException. Другие возможные исключения: ArgumentNullException (имя файла выражается нулевым значением), ArgumentException (параметр, задающий режим открытия файла, некорректен), SecurityException (пользователь не имеет права доступа) и DirectoryNotFoundException (указанный каталог не существует).

Ниже демонстрируется способ открытия файла test.dat для ввода данных:

```
FileStream fin;

try {
  fin = new FileStream("test.dat", FileMode.Open);
}
 catch(FileNotFoundException exc) {
   Console.WriteLine(exc.Message);
  return;
}
catch {
     Console.WriteLine("Невозможно открыть файл.");
  return;
}
```

Здесь первый оператор catch перехватывает ошибку, связанную с невозможностью обнаружения файла. Второй оператор catch обеспечивает перехват всех остальных исключений, связанных с выполнением файловых операций. Часто имеет смысл организовать выдачу сведений о каждой ошибке, что позволит располагать более точными сведениями о характере возникшего затруднения. Для простоты изложения в рассматриваемых примерах перехватывается лишь исключение FileNotFoundException.

Как к уже упоминалось, только что описанный конструктор FileStream открывается файл для чтения и записи. Если доступ необходимо ограничить только чтением или только записью, укажите строку следующего вида:

```
FileStream(string filename, FileMode mode, FileAccess how)
```

Как и прежде, параметр `filename` указывает имя открываемого файла, а параметр `mode` определяет режим открытия файла. Значение, передаваемое параметром `how`, задает способ доступа к файлу. Здесь может фигурировать одна из указанных ниже величин, заданных перечислением `FileAccess`:

```
FileAccess.Read FileAccess.Write FileAccess.ReadWrite
```

Например, следующая строка открывает файл только для чтения:

```
FileStream fin = new FileStream("test.dat", FileMode.Open, FileAccess.Read)
```

Если работа с файлом была завершена, необходимо закрыть его, используя метод `Close()`:

```
void Close()
```

При закрытии файла происходит освобождение системных ресурсов, выделенных для файла. С этого момента они становятся доступными при обработке другого файла. Метод `close()` может генерировать исключение `IOException`.

Чтение байтов с помощью класса `FileStream`

Класс `FileStream` определяет два метода, которые могут считывать байты из файла: `ReadByte()` и `Read()`. Для считывания одного байта из файла воспользуйтесь методом `ReadByte()`, общая форма которого приводится ниже:

```
int ReadByte()
```

Каждый раз при вызове этого метода из файла считывается один байт, который возвращается в виде целочисленного значения. Если при этом встречается признак конца файла, возвращается значение -1. В случае возникновения ошибки генерируется исключение `IOException`.

Для считывания блока байтов воспользуйтесь методом `Read()`:

```
int Read(byte[] buf, int offset, int numBytes)
```

Метод `Read()` предпринимает попытку считать байты, количество которых определяется параметром `numBytes`, из файла в буфер `buf`, начиная с позиции, заданной параметром `offset`. При этом возвращается количество успешно считанных байтов. В случае возникновения ошибки ввода/вывода генерируется исключение `IOException`. Возможны и другие исключения, в том числе исключение `NotSupportedException`, генерируемое в случае, когда операция чтения не поддерживается потоком.

В следующей программе используется метод `ReadByte()` для ввода и отображения содержимого текстового файла, имя которого указано в качестве аргумента командной строки. Обратите внимание на то, что блоки `try/catch` обрабатывают две ошибки, происходящие при первом запуске программы (указанный файл не найден либо имя файла не указано). Аналогичный подход может применяться всякий раз, когда используются аргументы командной строки.

```
/* Отображение файла.

   Для использования программы укажите имя файла,
   содержимое которого будет отображено.
   Например, для просмотра содержимого файла TEST.CS
   воспользуйтесь следующей командной строкой:

   ShowFile TEST.CS
*/
```

```
using System;
using System.IO;

class ShowFile {
  public static void Main(string[] args) {
    int i;
    FileStream fin;

    try {
      fin = new FileStream(args[0], FileMode.Open);
      } catch(FileNotFoundException exc) {
      Console.WriteLine(exc.Message);
      return;
    } catch(IndexOutOfRangeExceptton exc) {
      Console.WriteLine(exc.Message + "\nUsage: ShowFile File");
      return;
    }

    // Чтение байтов, пока не встретится символ EOF
    do {
      try {
        x = fin.ReadByte();
      } catch(IOException exc) {
        Console.WriteLine(exc.Message);
        return;
      }
      if(i != -1) Console.Write((char) i);
    } while(i != -1);

    fin.Close();
  }
}
```

Чтение из файла.

Если i равно -1, достигнут конец файла.

Запись в файл

Для записи байтов и файл воспользуйтесь методом `WriteByte()`:

```
void WriteByte(byte val)
```

Этот метод записывает байт, указанный параметром `val`, в файл. Если при записи возникла ошибка, генерируется исключение `IOException`. Если поток вывода не открыт, генерируется исключение `NotSupportedException`.

Байтовый массив может быть записан в файл с помощью метода `Write()`:

```
int Write (byte[] buf, int offset, int numBytes)
```

Ответы профессионала

Вопрос. Уже отмечалось, что метод `ReadByte()` возвращает значение -1 по достижении конца файла, но при этом отсутствует специальное возвращаемое значение в случае возникновения ошибки. Почему?

Ответ. В C# ошибки обрабатываются с помощью исключений. Поэтому, если метод `ReadByte()` либо другой метод ввода/вывода возвращает значение, это означает, что ошибка не произошла. При этом обеспечивается более «прозрачный» путь обработки ошибок ввода/вывода, чем в случае возврата специальных кодов ошибок.

Метод Write() записывает в файл количество байтов, указанное параметром numBytes, из массива buf, начиная с позиции, которая задана с помощью параметра offset. При этом возвращается количество записанных байтов. Если в процессе записи происходит ошибка, генерируется исключение IOException. Если поток вывода не открыт, генерируется исключение NoteSupportedException. Возможно возникновение и других исключении.

Наверное, вы знаете о том, что при выполнении вывода в файл часто данные не записываются сразу же на физическое устройство. Вместо этого вывод буферизуется операционной системой до тех пор, пока не накопится достаточно большой фрагмент данных, который сразу же записывается на диск. В результате возрастает производительность системы. Например, запись данных в файл на диске осуществляется по секторам, размеры которых варьируются от 128 байт и больше. Результаты вывода программы обычно буферизуются до тех пор, пока на диск не может быть записан целый сектор Однако если требуется, чтобы данные записывались на физическое устройство независимо от степени заполнения буфера, можно воспользоваться методом Flush():

```
void Flush()
```

Если при выполнении этой операции возникает ошибка, генерируется исключение IOException.

По завершении операции вывода файла необходимо его закрыть, воспользовавшись методом Close(). При этом гарантируется, что любые выводимые данные, находящиеся в буфере. Будут записаны в файл. Поэтому не стоит вызывать метод Flush() перед закрытием файла.

В следующем примере выполняемся копирование файла. Имена исходного и целевого файлов задаются в командной строке.

```
/* Копирование файла.

   При использовании этой программы укажите имена исходного
   и целевого файлов.
   Например, для кодирования файла FIRST.TXT
   в файл SECOND.TXT воспользуйтесь следующей
   командной строкой:

   CopyFile FIRST.TXT SECOND.TXT
 */

using System;
using System.IO;

class CopyFile {
  public static void Main(string[] args) {
    int i;
    FileStream fin;
    FileStream fount;

    try {
      // Открытие исходного файла
      try {
        fin = new FileStream(args[0], FileMode.Open);
      } catch(FileNotFoundException exc) {
```

```
        Console.WriteLine(exc.Message + "\nИсходный файл не найден");
        return;
      }

      // Сокрытие файла-копии
      try {
        fout = new FileStream(args[1], FileMode.Create);
      } carch(FileNotFoundException exc) {
        Console.WriteLine(exc.Message + "\nОшибка создания файла-копии");
        return;
      }
    } catch(IndexOutOfRangeException exc) {
      Console.WriteLine(exc.Message + "\nUsage: CopyFile From To");
      return;
    }

    // Копирование файла
    try {
      do {
        i = fin.ReadByte();
        if(i != -1) fout.WriteByte((byte)i);
      } while (i != -1);
    } catch(IoException exc) {
      Console.WriteLine(exc.Message + "Файловая ошибка");
    }

    fin.Close();
    fout.Close();
  }
```

Считывание байтов из одного файла с последующей их записью в другой файл.

Минутный практикум

1. Что возвращает метод `ReadByte()` по достижении конца файла?
2. Дли чего предназначен метод `Flush()`?
3. Какой метод следует вызвать для записи блока байтов?

Ввод/вывод в символьные файлы

Хотя существуют общие методы обработки байтовых файлов, для этой цели можно также воспользоваться символьными потоками. Преимущество символьных потоков проявляется в том, что они оперируют непосредственно с символами Unicode. Поэтому, если возникает потребность в хранении текста Unicode, наилучшим выбором в этом случае будут символьные потоки. Как правило, при выполнении файловых операций, основанных на использовании символов, класс `FileStream` включается и состав класса `StreamReader` или `StreamWriter`. Эти классы обеспечивают автоматическое преобразование байтовых потоков в символьные и наоборот.

1. По достижении конца файла метод `ReadByte()` возвращает значение -1.

2. Вызов метода `Flush()` приводит к тому, что данные, находящиеся в буферах вывода, физически записываются на устройство.

3. Для записи блока байтов вызывается метод `Write()`.

Помните о том, что на уровне операционной системы файл состоит из набора байтов. Это остается справедливым и для классов `StreamReader` или `StreamWriter`.

Класс `StreamWriter` наследуется из класса `TextWriter`. Класс `StreamReader` наследуется из класса `TextReader`. В результате классы `StreamWriter` и `StreamReader` получают доступ к методам и свойствам, определенным наследуемыми ими классами.

Использование класса `StreamWriter`

Для создания символьного потока вывода включите объект `Stream` (например, `FileStream`) в состав класса `StreamWriter`. Класс `StreamWriter` определяет несколько конструкторов. Один из наиболее популярных конструкторов показан ниже?

```
StreamWriter(Stream stream)
```

Здесь параметр `stream` указывает имя открытого потока. Если указанный поток будет пуст, конструктор вызывает исключение `ArgumentException`, а если `stream` равен нулю, вызывается исключение `ArgumentNullException`. Сразу же после создания класс `StreamWriter` автоматически выполняет преобразование символов в байты.

Ниже приводится код простой утилиты, выполняющей запись информации на диск, которая считывает строки текста, введенного с клавиатуры, а затем записывает его в файл `rest.txt`. Считывание текста продолжается до тех пор, пока пользователь не введет слово «stop». Утилита использует класс `FileStream`, включенный в состав класса `StreamWriter`, с целью вывода информации в файл.

```
/* Простая утилита, выполняющая запись на диск,
   которая демонстрирует возможности класса StreamWriter. */

using System;
using System.IO;

class KtoD {
  public static void Main() {
    string str;
    FileStream fout;

    try {
      fout = new FileStream("test.txt", FileMode.Create);
    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message + "Невозможно открыть файл.");
      return;
    }
    StreamWriter fstr_out = new StreamWriter(fout);

    Console.WriteLine("Введите текст stop для выхода.");
    do {
      Console.Write (" : ");
      str = Console.ReadLine();

      if(str != "stop") {
        str = str + "\r\n"; // Добавление символов образования новой строки.
        try {
          fstr_out.Write(str);
```

Создание класса `StreamWriter`.

Запись строк в файл.

```
        } catch(IOException exc) {
          Console.WriteLine(exc.Message + "Файловая ошибка");
          return;
        }
      }
      while(str != "stop");

    fstr_out.Close();
  }
}
```

В некоторых случаях можно открыть файл напрямую, используя класс `StreamWrite`. При этом используется один из указанных ниже конструкторов:

```
StreamWriter(string filename)

Stream.Writer(string filename, bool appendFlag)
```

Здесь параметр `filename` указывает имя открываемого файла, причем это может быть полностью определенное имя. При использовании конструктора второго вида данные будут добавляться в конец существующего файла, если флагу `appendTrue` присвоено значение `true`. Если же упомянутому флагу присвоено значение `false`, данные будут замещать содержимое указанного файла. При отсутствии файла в обоих случаях происходит его создание. Также в обоих случаях генерируется исключение `IOException` в случае наличия ошибки. Могут генерироваться и другие исключения.

Ниже приводится код утилиты, выполняющей запись информации на диск. Код переписан таким образом, что включает класс `StreamWriter`, выполняющий открытие выходного файла.

```
/* Открытие файла с помощью класса StreamWriter. */

using System;
 using System.IO;

class KtoD {
  public static void Main() {
    string str;
    StreamWriter fstr_out;

    try {
      fstr_out = new StreamWriter("test.txt");
    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message + "Невозможно открыть файл.");
      return ;
    }

    Console.WriteLine("Введите текст 'stop' для выхода.");
    do {
      Console.Write(" : ");
      str = Console.ReadLine();

      if(str != "stop") {
        str = str + "\r\n"; // Добавление символов образования новой строки.
        try {
          fstr__out.Write(str);
        } catch(IOException exc) {
```

```
            Console.WriteLine(exc.Message + "Файловая ошибка");
            return ;
          }
        }
    } while(str != "stop");

    fstr_out.Close();
  }
}
```

Использование класса `StreamReader`

Для создания символьного потока ввода включите байтовый поток в класс `StreamReader`. Этот класс определяет несколько конструкторов. Наиболее часто используется следующий:

```
StreamReader(Stream stream)
```

Здесь параметр `stream` определяет имя открытого потока. Если значение параметра `stream` равно нулю, генерируется исключение `ArgumentNullException`. Сразу же после создания класс `StreamReader` будет автоматически выполнять преобразование байтов в символы.

Следующая программа представляет собой простую утилиту; она считывает текстовый файл `test.txt` и отображает его содержимое на экране. Таким образом, данная утилита является комплементарной по отношению к утилите записи на диск, приведенной в предыдущем разделе.

```
/* Простая утилиза, которая демонстрирует возможности класса FileReader. */

using System;
using System.IO;

class DtoS {
  public static void Main() {
    FileStream fin;
    string s;

    try {
      fin = new FileStream("tes.txt", FileMode.Open);
    }
    catch(IOExceptxon exc) {
      Console.WriteLine(exc.Message - "Невозможно открыть файл.");
      return ;
    }

    StreamReader fstr_in = new StreamReader(fin);

    while((s = fstr_in.ReadLine()) != null) {
      Console.WriteLine(s);
    }

    fstr_in.Close();
  }
}
```

Считывание строк из файла с последующим их отображением на экране.

Обратите внимание на то, каким образом определяется конец файла. Как только ссылка, возвращаемая методом `ReadLine()`, получит значение, равное нулю, это укажет на достижение конца файла.

Как и в случае с классом `StreamWriter`, в некоторых случаях можно открыть файл непосредственно, воспользовавшись возможностями класса `StreamReader`. Для выполнения этой операции служит следующий конструктор:

```
StreamReader(string filename)
```

Здесь параметр `filename` указывает имя открываемого файла, причем может задаваться полный путь. Существование файла является обязательным. Если файл не существует генерируется исключение `IOException`. Если параметр `filename` представляет собой пустую строку, генерируется исключение `ArgunentException`.

Минутный практикум

1. Какой класс используется для считывания символов из файла?
2. Какой класс применяется для записи символов в файл?
3. Зачем нужны отдельные символьные классы ввода/вывода?

Перенаправление стандартных потоков

Как упоминалось ранее, стандартные потоки, такие как `Console.In`, могут быть перенаправлены. Конечно, наиболее часто потоки перенаправляются в файл. Как только стандартный поток перенаправлен, «водимые и выводимые данные перенаправляются автоматически. Путем перенаправления стандартных потоков программа может считывать команды из файла на диске, создавать журнальные файлы либо даже получать данные из сети.

Ответы профессионала

Вопрос. Я слышал, что можно указать «символьная кодировка» путем открытия класса `StreamReader` или `StreamWriter`. В чем суть символьного кодирования и как его можно использовать?

Ответ. Классы `ScreamReader` и `StreamWriter` преобразуют байты в символы и наоборот. При этом используется *символьная кодировка,* указывающая способ преобразования. По умолчанию в C# используется кодировка UTF-8, которая является совместимой с кодировкой Unicode. Поскольку ASCII является подмножеством Unicode, свойство кодировки, заданное по умолчанию, обрабатывает символы ASCII и Unicode. Для указания другой кодировки можно воспользоваться перегруженными версиями конструкторов `StreamReader` или `StreamWriter`, которые включают параметр кодирования. Иначе говоря, символьная кодировки требуется довольно редко.

---

1. Для считывания символов используется класс `StreamReader`.
2. Для записи символов применяется класс `StreamWriter`.
3. Символьные классы ввода/вывода автоматизируют прямое и обратное преобразование `byte-char`.

Перенаправление стандартных потоков выполняется двумя путями. Во-первых, при вызове программы в командной строке можно воспользоваться символами > и < выполняющими перенаправление в Console.In и в Console.Out, соответственно. Например, обратите внимание на такую программу:

```
using System;

class Test {
  public static void Main()   {
    Console.WriteLine("Это текст.");
   }
 }
```

При вызове программы следующим образом:

```
Test > log
```

строка "Это текст." записывается в файл log. Ввод может перенаправляться таким же образом. Единственное, что нужно помнить при направлении ввода, — убедиться в том, что источник входной информации передает количество входных данных, достаточное для удовлетворения требований программы. Если же это требование не выполняется, программа «зависает».

Операторы перенаправления командной строки (< и >) не являются частью C#, но они поддерживаются операционной системой. Таким образом, если в среде поддерживается перенаправление операций ввода/вывода (как в случае с Windows), можно перенаправлять стандартный ввод и вывод без выполнения каких-либо изменений в программе. Однако существует и второй способ перенаправления стандартных потоков, при использовании которого применяется программный контроль. В этом случае необходимо воспользоваться методами SetIn(), SetOut() и SetError(), которые являются членами класса Console:

```
static void SetIn(TextReader input)

static void SetOut(TextWriter output)

static void SetError(TextWriter output)
```

Таким образом, для перенаправления вывода вызовите метод SetIn(), указав требуемый поток. Можно использовать любой поток ввода в том случае, если он наследуется из класса TestReader. Для перенаправления вывода в файл укажите файл, который входит в состав класса TextWriter. Следующая программа демонстрирует соответствующий пример.

```
// Перенаправление Console.Out.

using System;
using System.IO;

class Redirect {
  public static void Main() {
    StreamWriter log_out;

    try {
      log_out = new StreamWriter("logfile.txt");
    }
    catch(IOException exc) {
```

```
      Console.WnteLine(exc.Message + "Невозможно открыть файл.");
      return ;
    }

    Console.SetOut(log_out);
    Console.WriteLine("Начало журнального файла.");

    for(int i=0; i<10; i++) Console .WriteLine(i);

    Console.WriteLme("Конец журнального файла.");
    log_out.Close();
  }
}
```

Перенаправление потока `Console.Out`.

После запуска программы на выполнение на экране ничего не отображается. Весь вывод направляется в файл `logfile.txt`:

```
Начало журнального файла.
0
1
2
3
4
5
5
7
8
9
Конец журнального файла.
```

Теперь вы можете поэкспериментировать с другими встроенными потоками.

Проект 11-1. Утилита сравнения файлов

`CompFiles.cs`

В этом проекте разрабатывается простая, но весьма полезная утилита, используемая при сравнении файлов. Эта утилита открывает оба сравниваемых файла, а затем считывает и сравнивает каждый соответствующий набор байтов. Если при этом обнаруживается несоответствие, файлы считаются различными. Если одновременно Достигнуты концы обоих файлов и несоответствие не найдено, файлы считаются одинаковыми.

Пошаговая инструкция

1. Создайте файл `CompFiles.cs`.
2. В файл `CompFiles.cs` введите следующий код:

```
/*
   Проект 11—1.
   Сравнение двух файлов.
   Укажите имена двух сравниваемых файлов в командной строке.
   Например:
        CompFile FIRST.TXT SECOND.TXT
        */
```

```
using System;
using System.IO;

class CompFiles {
  public static void Main(string[] args) {
    ini i=0, j=0;
    FileStream f1;
    FileStream f2;

    try {
      // Открытии первого файла.
      try {
       f1 = new FileStream(arqs[0], FileMode.Open);
      } catch(FileNotFoundException exc) {
        Console.WriteLine(exc.Message);
        return;
      }

      // Открытие второго файла.
      try {
        f2 = new FileStream(args[1], FileMode.Open);
      } catch(FileNotFoundException exc) {
        Console.WriteLine(exc.Message);
        return;
      }
    } catch(IndexOutofRangeException exc) {
      Console.WriteLine(exc.Message + "\nUsage: CompFile fl f2");
      return;
    }

    // Сравнение файлов.
    try {
      do {
        i = f1.ReadByte();
        j = f2.ReadBvte();
        it(i != j) break;
      } while (i != -1 && j != -1);
    } catch(IOException exc) {
      ConsoLe.WriteLine(exc.Message);
    }
    if (i != j)
      Console.WriteLine("Файлы различны.");
    else
      Console.WriteLine("Файны совпадают.");

    f1.Close();
    f2.Close();
  }
}
```

3. С целью тестирования программы `CompFiles` скопируйте файл `CompFiles.cs` в файл `temp`. Затем выполните следующую команду:

```
CompFiles CompFiles.cs temp
```

Программа должна сообщить о том, что файлы совпадают.

4. Затем сравните файлы CompFiles.cs и CopyFiles.cs (показан ранее), воспользовавшись следующей командой:

```
CompFiles CompFiles.cs CopyFile.cs
```

5. Программу CompFiles можно обогатить различными опциями. Например, можно добавить опцию, позволяющую программе игнорировать регистр символов. Другое расширение ее возможностей заключается в том, что программа CompFiles могла бы отображать позиции отличающихся данных в файлах.

Чтение и запись двоичных данных

До сих пор мы рассматривали механизмы чтения и записи байтов и символов, но следует учитывать, что можно производить чтение и запись данных других типов, например значений типа int, double или short. Для чтения и записи двоичных значений, имеющих встроенные типы данных C#, применяются потоки BinaryReader и BinaryWriter. При использовании этих потоков важно понимать, что данные считываются и записываются во внутреннем двоичном формате, а не в текстовой форме, воспринимаемой человеком.

Поток BinaryWriter

Поток BinaryWriter представляет собой оболочку для байтового потока, которая управляет записью двоичных данных. Ниже показан наиболее часто используемый конструктор:

```
BinaryWriter(Stream outputStream)
```

Здесь параметр outputStream определяет поток, в который производится запись данных. Для записи вывода в файл в качестве этого параметра указывается объект, созданный конструктором FileStream. Если значение параметра outputStream равно нулю, генерируется исключение ArgumentNuliException. Если параметр outputStream задаст поток, не открытый для записи, генерируется исключение ArgumentException.

Поток BinaryWriter определяет методы, которые обеспечивают запись данных всех встроенных типов C#. Многие из этих методов приведены в табл. 11.5. Поток BinaryWriter также определяет стандартные методы Close() и Flush(), работа которых описывалась ранее.

Таблица 11.5. Наиболее часто используемые методы, определенные потоком BinaryWriter

Метод	Описание
void Write(sbyte ***val***)	Запись байта со знаком
void Write(byte ***val***)	Запись байта без знака
void Write(byte[] ***but***)	Запись массива байтов
void Write(short ***val***)	Запись короткого целого
void Write(ushort ***val***)	Запись короткого целого без знака
void Write(int ***val***)	Запись целого числа
void Write (uint ***val***)	Запись целого без знака

Метод	Описание
`void Write (long val)`	Запись длинного целого
`void Write (ulong val)`	Запись длинного целого без знака
`void Write (float val)`	Запись значения типа `float`
`void write (double val)`	Запись значения типа `double`
`void Write (char val)`	Запись символа
`void Write (char[] val)`	Запись массива символов
`void Write (string val)`	Запись строки

Поток `BinaryReader`

Поток `BinaryReader` является оболочкой, которая включает байтовый поток, выполняющий чтение двоичных данных. Ниже показан наиболее часто используемый конструктор для этого потока.

```
BinaryReader(Stream inputStream)
```

Здесь параметр `inputStream` обозначает поток, из которого происходит считывание данных. Для выполнения считывания данных из файла в качестве этого параметре можно указать объект, созданный с помощью конструктора `FileStream`. Если значение параметра `inputStream` равен нулю, генерируется исключение `ArgumentNullException`. Если параметр `inputStream` не открыт для чтения, генерируется исключение `ArgumentException`.

Поток `BinaryReader` поддерживает методы, обеспечивающие считывание всех простых типов данных в C#. Наиболее часто используемые из них приведены в табл. 11.6. Поток `BinaryReader` также определяет три версии метода `Read()`, приведенные ниже.

`int Read()`	Возвращает целочисленное представление следующего доступного символа из потока ввода. При достижении конца, файла возвращает значение -1
`int Read (byte[] buf, int offset, int num)`	Осуществляет попытку считать *num* байтов в буфер *buf,* начиная с элемента, находящегося в позиции *offset*, возвращая при этом количество успешно считанных байтов
`int Read (char[] buf, int offset, int num)`	Осуществляет попытку считать *num* символов в буфер *buf,* начиная с элемента, находящегося в позиции *offset*, возвращая при этом количество успешно считанных символов

В результате сбоя при выполнении этих методов генерируется исключение `IOException`.

Также определен стандартный метод `Close()`.

Таблица 11.6. Наиболее часто используемые методы, определенные потоком `BinaryReader`

Метод	Описание
`bool ReadBooLean()`	Чтение значения типа `bool`
`byte ReadByte()`	Чтение значения типа `byte`
`sbyte ReadSByte()`	Чтение значения типа `sbyte`
`byte[] ReadBytes(int num)`	Чтение *num* байтов и возврат их в виде массива
`char ReadChar()`	Чтение значения типа `char`
`char[] ReadChars(int, num)`	Чтение *num* символов и возврат их в виде массива
`double ReadDouble()`	Чтение значения типа `double`
`float ReadSingie()`	Чтение значения типа `float`
`short ReadInt16()`	Чтение значения типа `short`
`int ReadInt32()`	Чтение значения типа `int`
`long Readlnt64()`	Чтение значения типа `long`
`ushort ReadUint16()`	Чтение значения типа `ushort`
`uint ReadUint32()`	Чтение значения типа `uint`
`ulong ReadUint64()`	Чтение значения типа `ulong`
`string ReadString()`	Чтение строки

Демонстрация двоичного ввода/вывода

Ниже приводится пример программы, которая демонстрирует возможности потоков `BinaryReader` и `BinaryWriter`. Эта программа сначала записывает в файл, а затем считывает из него значения различных типов.

```
// Запись и считывание двоичных данных.
using System;
using System.IO;

class RWData {
  public static void Main() {
    BinaryWriter dataOut;
    BinaryReader dataIn;

    int i = 10;
    double d = 1023.56;
    bool b = true;

    try {
      dataOut = new
        BinaryWriter(new FileStream("testdata", FileMode.Create));
     }
    catch(IOException exc) {
      Console.WriteLine(exc.Message + "\nНевозможно открыть файл.");
      return;
    }

    try {
      Console.WriteLine("Запись " + i);
```

```
      dataOut.Write(i);

      Console.WriteLine("Запись " + d);
      datdOut.Write(d);

      Console.WriteLine("Запись " + b);
      dataOut.Write(b);

      Console.Writeline("Запись " + 12.2 * 7.4);
      dataOut.Write(12.2 * 7.4);

    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message + "\nОшибка записи.");
    }

    dataOut.Close();

    Console.WriteLine();

    // Теперь снова чтение.
    try {
      dataIn = new
           BinaryReader(new FileStream("testdata", FileMode.Open));
    }
    catch(IOException exc) {
      Console.Writeline(exc.Message + "\nНевозможно открыть файл.");
      return;
    }

    try {
      i = dataIn.ReadInt32();
      Console.WriteLine("Чтение " + i);

      d = dataIn.ReadDouble();
      Console . Wri teLine ("Чтение " + d);

      b = dataIn.ReadBoolean();
      Console.WriteLine("Чтение " + b);

      d = dataIn.ReadDoubie();
      Console.WriteLine("Чтение " + d);
    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message + "Ошибка чтения.");
    }

    dataIn.Close();
  }
}
```

Запись двоичных данных.

Чтение двоичных данных.

Результат выполнения программы:

```
Запись 10
Запись 1023.56
Запись True
```

```
Запись 90.28

Чтение  10
        Чтение 1023.56
  Чтение True
        Чтение 90.23
```

Минутный практикум

1. Какой поток используется при записи в двоичные файлы?
2. Какой метод вызывается для записи значения типа `double` в двоичном формате?
3. Какой метод вызывается для чтения значения типа `short` в двоичном формате?

Произвольный доступ к содержимому файл

До настоящего момента времени нами использовался так называемый последовательный доступ к содержимому файла, байт за байтом. Однако в C# возможно произвольный доступ к содержимому файлов. Для реализации подобного действия используется метод `Seek()`, определенный в классе `FileStream`. Этот метод позволяет задавать индикатор позиции текущего элемента данных в файле (который называется файловый указатель).

Синтаксис метода `Seek()` таков:

```
long Seek(long newPos, SeekOrigin origin)
```

Здесь параметр `newPos` указывает на новую позицию (в байтах) для файлового указания начиная от местоположения, указанного параметром `origin`. Последний параметр может иметь одно из значений, определенных в перечислении `SeekOrigin`.

Значение	Описание
`Begin`	Отсчет от начала файла
`Current`	Отсчет от текущего местоположения
`End`	Отсчет от конца файла

После вызова метода `Seek()` следующие операции чтения либо записи производится в новой позиции файлового указателя. Если во время позиционирования указателя произойдет ошибка, генерируется исключение `IOException`. Если поток не ____ живает операцию позиционирования, генерируется исключение `NotSupportException`.

Ниже приводится пример, демонстрирующий способ выполнения операции ввода/ввывода с произвольным доступом к содержимому файла. В этом примере не помещаются прописные буквы алфавита, которые затем считываются в производном порядке.

---

1. При записи в двоичные файлы используется поток `BinaryWriter`.
2. Для записи значения типа `double` вызывается метод `Write()`.
3. Для чтения значения типа `short` вызывается метод `readInt16()`.

```
// Демонстрация механизма произвольного доступа к содержимому файла.
using System;
using System.IO;

class RandomAccessDemo {
  public static void Main() {
    FileStream f;
    char ch;

    try {
      f = new FileStream("random.dat", FileMode.Create);
    }
    catch(FileNotFoundException exc) {
      Console.WnteLine(exc.Message);
      return ;
    }

    // Запись букв в алфавитном порядке.
    for(int i=0; i < 26; i++) {
      try {
        f.WriteByte((byte)('A'+i));
      }
      catch(IOException exc) {
        Console.WriteLine(exc.Message);
        return ;
      }
    }

    try {
      // Считывание букв.
      f.Seek(0, SeekOrigin.Begin); // Поиск первого байта.
      ch = (char) f.ReadByte();
      Console.WriteLine("Первое значение " + ch);

      f.Seek(1, SeekOrigin.Begin); // Поиск второго байта.
      ch = (char) f.ReadByte();
      Console.WriteLine("Второе значение " + ch);

      f.Seek(4, SeekOrigin.Begin); // Поиск пятого байта.
      ch = (char) f.ReadByte();
      Console.WriteLine(Пятое значение " + ch);

      Console.WriteLine();

      // Теперь чтение четных букв.
      Console.WriteLine("Четные буквы: ");
      for(int i=0; i < 26; i += 2) {
        f.Seek(i, SeekOrigin.Begin); // Поиск i-го байта
        ch = (char) f.ReadByte();
        Console.Write(ch + " ");
      }
    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message);
    }

    f.Close();
  }
}
```

Результат выполнения программы:

```
       Первое значение A
Второе значение B
Пятое значение E

      Здесь каждое другое значение:
A C E G I K M O Q S U W Y
```

Минутный практикум

1. Каким образом позиционируется файловый указатель?
2. Какое значение параметра `origin` задает отсчет от текущего значения файлового указателя?
3. Какое исключение генерируется, если метод `Seek()` вызывается для потока, который не поддерживает позиционирование?

Преобразование числовых строк в их внутренние представления

Перед завершением рассмотрения механизмов ввода/вывода обратимся к технике, которая используется при чтении числовых строк. Как вы помните, метод `WriteLine()` в C# обеспечивает удобный способ вывода данных различных типов на консоль. Сюда также включаются числовые значения встроенных типов данных, таких как int и `double`. В результате метод `WriteLine()` автоматически преобразует числовые значения в их представления, более удобные для восприятия человеком. Однако в C# не поддерживается метод ввода, который мог бы считывать и преобразовывать строки, содержащие числовые значения, в их внутреннее двоичное представление. Например, невозможно ввести с клавиатуры такую строку, как «100» и автоматически преобразовать ее в соответствующее целочисленное значение, которое может быть сохранено в переменной типа `int`. Для выполнения этой задачи необходимо воспользоваться методом, который определен для всех встроенных числовых типов: `Parse()`.

Перед тем как перейти к дальнейшему рассмотрению вопроса, необходимо вспомнить один важный факт: все встроенные типы C#, такие как `int` и `double`, на самом деле являются псевдонимами (другими именами) для структур, определенных в библиотеке .NET Framework. Фактически Microsoft утверждает, что тип языка C# и тип структуры .NET практически совпадают. Просто используются различные наименовании. Поскольку типы значений C# поддерживаются структурами, то эти типы являются членами соответствующих структур.

Ниже показаны типы числовых значений C# и соответствующие им имена структур .NET

---

1. Для позиционирования файлового указателя используется метод `Seek()`.
2. Значение `SeekOrigi.Current`.
3. Если метод `Seek()` не поддерживается потоком, генерируется исключение `NotSupportedException`.

Имя структуры .NET	Тип данных C#
Double	double
Single	float
Intl6	short
Int32	mt
Int64	long
UIntl6	ushort
UInt32	uint
UInt64	ulong
Byte	byte
Sbyte	sbyte

Эти структуры определены в пространстве имен System. Таким образом, полностью определенное имя структуры Int32 будет System.Int32. Эти структуры предлагают широкий набор методов, позволяющих полностью интегрировать типы значений в иерархию объектов C#. Также числовые структуры определяют статические методы,, преобразующие числовую строку в соответствующий ей двоичный эквивалент. Эти методы приведены ниже. Каждый из них возвращает двоичное значение, соответствующее строке.

Структура	Метод для преобразования
Double	static double Parse(string ***str***)
Single	static float Parse(string ***str***)
Int64	static long Parse(string ***str***)
Int32	static int Parse(string ***str***)
Int16	static short Parse(string ***str***)
UInt64	static ulong Parse(string ***str***)
UInt32	static uint Parse(string ***str***)
UIntl6	static ushort Parse(string ***str***)
Byte	static byte Parse(string ***str***)
SByte	static sbyte Parse(string ***str***)

Методы Parse() генерируют исключение FormatException, если параметр str не содержит корректного значения, соответствующего нужному типу данных. Исключение ArgumentNullException генерируется, если параметр str имеет значение null, а если строка str после преобразования не может быть помещена в переменную соответствующего типа, генерируется исключение OverflowException.

Методы Parse() обеспечивают удобный способ преобразования в числовое значение строки, считываемой с клавиатуры или из текстового файла, в подходящий внутренний формат. Например, в следующей программе подсчитывается среднее значение чисел, введенных пользователем. Сначала программа просит указать количество значений, для которых рассчитывается среднее значение. Затем она считывает это число с помощью метода ReadLine и посредством метода Int32.Farse() преобразовывает строку к целочисленное значение. Далее осуществляется ввод значений, причем метод Double.Parse() используется для преобразования строк в их числовые эквиваленты, имеющие тип double.

```
/* Эта программа подсчитывает среднее значение введенных чисел */

using System;
using System.IO;

class AvgNums {
  public static void Main() {
    string str;
    int n;
    double sum = 0.0;
    double avg, t;

    Console.Write("Укажите количество вводимых чисел: ");
     str = Console.ReadLine();
    try {
      n = Int32.Parse(str);
    }
    catch(FormatException exc) {
      Console.WriteLine(exc.Message);
      n = 0;
    }
    catch(OverflowException exc) {
      Console.WriteLine(exc.Message);
      n = 0;
    }

    Console.WriteLine("Введите " + n + " значений.");
    for(int i=0; i < n ; i++) {
      Console.Write(": ");
      str = Console.ReadLine();
      try {
        t = Double.Parse(str);
      } catch(FormatException exc) {
        Console.WriteLine(exc.Message);
        t = 0.0;
      }
      catch(OverflowException exc) {
        Console.WriteLine(exc.Message);
        t = 0;
      }
      sum += t;
    }
    avg = sum / n;
    Console.WriteLine("Среднее значение " + avg);
  }
}
```

Преобразование строки в число типа `int`.

Преобразование строки в число типа `double`.

Результат выполнения программы:

```
Укажите количество вводимых чисел: 5
Введите 5 значений.

: 2.2
: 3.3
: 4.4
: 5.5
Среднее значение 3.3
```

С помощью методов Parse() можно улучшить калькулятор платежей по ссуде рассматриваемый в проекте 2-3. В старой версии величина ссуды, значения процентной ставки, а также некоторые другие значения жестко задавались в самой программе. Программа станет более универсальной, если эти значения будут определяться пользователем. Ниже приводится код усовершенствованного калькулятора платежей по ссуде.

```
/*
   Улучшенный проект 2—3

   Вычисление регулярных платежей по ссуде.
*/

using System;

class RegPay {
  public static void Main() {
    decimal Principal; //        Величина ссуды
    decimal IntRate; //          Процентная ставка в виде десятичной дроби
                              // (например, 0.075)
    decimal PayPerYear; // Количество платежей в год
    decimal NumYears; // Количество лет
    decimal Payment; // Размер платежа
    decimal numer, denom; // Временные переменные
    double b, e; // Параметры вызова метода Pow()

    string str;

    Console.Write("Введите величину ссуды: ");
    str = Console.ReadLine();
    try {
      Principal = Decimal.Parse(str);
    } catch(FormatException exc) {
      Console.WriteLine(exc.Message);
      return;
    }

    Console.Write("Введите процентную ставку (в виде 0.085): ");
    str = Console.ReadLine();
    try {
      IntRate = Decimal.Parse(str);
    } catch(FormatException exc) {
      Console.WriteLine(exc.Message);
      return;
    }

    Console.Write("Введите количество лет: ");
    str = Console.ReadLine();
    try {
      NumYears = Decimal.Parse(str);
    } catch(FormatException exc) {
      Console.WriteLine(exc.Message);
      return;
    }

    Console.Write("Введите количество платежей в год: ");
```

```
    str = Console.ReadLine();
    try {
      PayPerYear = Decimal.Parse(str);
    } catch(FormatException exc) {
      Console.WriteLine(exc.Message);
      return;
    }

    numer = IntRate * Principal / PayPerYear;

    e = (double) - (PayPerYear * NumYears);
    b = (double) (IntRate / PayPerYear) + 1;

    denom = 1 - (decimal) Math.Pow(b, e);

    Payment = numer / denom;

    Console.WriteLine("Платеж {0:C}", Payment);
  }
}
```

Результат выполнения программы:

```
Введите величину ссуды: 10000
Введите процентную ставку (в виде 0.085): 0.075
Введите количество лет: 6
Введите количество платежей в год: 12
Платеж $200.38
```

Ответы профессионала

Вопрос. Для чего еще пригодны структуры значение-тип, такие как `Int32` или `Double`?

Ответ. Структуры значение-тип поддерживают несколько методов, облегчающих интеграцию встроенных типов C# в иерархию объектов. Например, все структуры обладают методами `CompareTo()`, которые используются для сравнения значении. методами `Equals()`, которые проверяют два значения на равенство; а также методами, которые возвращают значение объекта в различных формах. Числовые структуры также включают поля `MinValue` и `MaxValue`, содержащие максимальное и минимальное значения, которые могут быть сохранены посредством объекта данного типа.

Проект 11-2. Создание справочной системы

`FileHelp.cs`

В проекте 4.1 был создан класс `Help`, выдающий сведения об управляющих операторах C#. В подобной реализации справочная информация хранится в классе, а пользователь может выбирать ее в меню, воспользовавшись нумерованными опциями. Хотя этот подход является вполне функциональным, вообще говоря, он не идеален при создании справочной системы. Например, если понадобится изменить справочную информацию, придется вносить изменения в исходный код программы. Кроме того, выбор тем справки по номерам является более утомительным, чем поиск

по имени, и неприемлем в случае обширных списков тем. Поэтому этот недостаток был устранен в следующей версии справочной системы.

Справочная система хранит информацию в файле справки. Этот файл представляем собой стандартный текстовый файл, который можно изменять и расширять по желанию пользователя, не прибегая при этом к изменению самого кода программы, Пользователь может получать нужную ему информацию путем ввода имени желаемой темы. После этого справочная система производит поиск согласно введенному ключевому слову. Если соответствующая информация найдена, она отображается на экране.

Пошаговая инструкция

1. Сначала потребуется создать файл, который будет использоваться справочной системой. Этот файл является стандартным текстовым файлом, который может быть организован следующим образом:

```
#имя-темы1
справка по теме 1

#имя-темы2
справка по теме 2

 .
 .
 .

#имя-темыN
справка по теме N
```

Имя каждой темы должно находиться в отдельной строке и предваряться символом #. Благодаря использованию знака # программа может быстро находить начало каждой темы. После имени темы может указываться произвольное количество строк справочной информации, посвященной этой теме. При этом нужно позаботиться о пустой строке между последним предложением справки по одной теме и началом следующей. После конца каждой строки не требуются завершающие пробелы.

Ниже приводится простой пример справочного файла, который может использоваться дая тестирования справочной системы. Этот файл хранит сведения об управляющих операторах C#.

```
if (condition) statement;
else statement;

#switch
switch(expression) {
  case constant:
    statement sequence
    break;
    // ...
  }

#for
```

```
for(init; condition; iteration) statement;

#while
 while(condition) statement;

#do
do {
  statement;
  } while (condition);

#break
break; or break label;

#continue
continue; or continue label;
```

Назовите этот файл `helpfile.txt`.

2. Создайте файл `FileHelp.cs`.
3. Начните создавать новый класс `Help`, воспользовавшись следующими строками кода:

```
class Help {
  string helpfile; // Имя справочного файла.

  public Help(string fname) {
    helpfile = fname;
   }
```

Имя справочного файла передается конструктору `Help` и хранится в экземпляре переменной `helpfile`. Поскольку каждый экземпляр класса `Help` может содержать свою собственную копию файла `helpfile`, каждый экземпляр может предоставлять справку по отдельной теме.

4. Добавьте метод `helpon()`, код которого приведен ниже, в класс `Help`. Этот метод используется для выборки справки по указанной теме.

```
// Отображение справки по теме.
public bool helpon(string what) {
   StreamReader helpRdr;
   int ch;
   string topic, info;

   try {
     helpRdr = new StreamReader(helpfile);
   }
   catch(FileNotFoundException exc) {
     Console.WriteLine(exc.Message);
     return false;
   }

   try {
     do {
       // Считывание символов, пока не будет найден символ #
       ch = helpRdr.Read();

       // Проверка найденной темы на соответствие искомой
       if(ch == '#') {
         topic = helpRdr.ReadLine();
```

```
            if (what == topic) { // Найдены темы.
              do {
                 info = helpRdr.ReadLine();
                 if(info != null) Console.WriteLine(info);
               } while((info != null) && (info != ""));
              helpRdr.Close();
              return true;
            }
          }
} while(ch != -1);

    catch(IOException exc) {
      Console.WriteLine(exc.Message);
    }
    helpRdr.Close();
    return false; // Тема не найдена
}
```

Класс `Help` открывается с помощью метода `StreamReader`. Поскольку справочный файл содержит текст, за счет использования символьного потока справочная система обеспечивает поддержку различных языков.

Метод `helpon()` работает следующим образом. Строка, содержащая имя темы, передается в параметре `what`. Затем открывается справочный файл. После этого метод производит поиск в файле, сравнивая значение параметра `what` и наименование темы, находящейся в файле. Помните, что каждая тема начинается знаком #, поэтому цикл поиска просматривает файл на предмет наличия символов #. Как только очередная тема будет найдена, производится проверка на предмет того, соответствует ли тема, обозначенная знаком #, теме, соответствующей ключевому слову `what`. Если соответствие установлено, отображается информация, связанная с этой темой. В случае обнаружения соответствия метод `helpon()` возвращает значение `true`. Если же соответствие не установлено, он возвращает значение `false`.

5. Класс `Help` также поддерживает метод `getSelection()`:

```
// Получение справки по указанной теме.
  public string getSelection()     {
    string topic = "";

    Console.Write("Введите тему: ");
    try {
      topic = Console.ReadLine();
    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message);
      return "";
    }
    return topic;
  }
```

6. Ниже приводится полный код справочной системы.

```
/*
   Проект 11-2

   Справочная система использует файл на диске
   для хранения текстов справки.
*/
```

```
using System;
using System.IO;

/* Класс Help открывает справочный файл,
   ищет тему, затем отображает информацию
   по этой теме. */
class Help {
  string helpfile; // Имя справочного файла.

  public Help(string fname) {
    helpfile = fname;
  }

  // Отображение справки по теме.
  public bool helpon(string what) {
    StreamReader helpRdr;
    int ch;
    string topic, info;

    try {
      helpRdr = new StreamReader(helpfile);
    }
    catch(FileNotFoundException exc) {
      Console.WriteLine(exc.Message);
      return false;
    }

    try {
      do {
        // Считывание символов, пока не будет найден символ #
        ch = helpRdr.Read();

        // Сравнение тем.
        if(ch == '#') {
          topic = helpRdr.ReadLine();
          if(what == topic) { // Тема найдена
            do {
              info = helpRdr.ReadLine();
              if(info null) Console.WriteLine(info);
            } while((info != null) && (info !- ""));
            helpRdr.Close();
            return true;
          }

      } while(ch != -1);
    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message);
    }
    helpRdr.Close();
    return false; // Тема не найдена.
  }

  // Получение справки по указанной пользователем теме.
  public string getSeleccion() {
    string topic = "";

    Console.Write("Введите тему: ");
    try {
      topic = Console.ReadLine();
```

```
      }
      catch(lOException exc) {
        Console.WriteLine(exc.Message);
        return
      }
      return topic;
    }
  }

  // Демонстрация возможностей файловой справочной системы.
  class FileHelp {
    public static void Main() {
      Help hlpobj = new Help("helpfile.txt");
      string topic;

      Console.WriteLine("Проверка справочной системы. " +
                        "Введите stop для выхода.");
      do {
        topic = hlpobj.getSelection();

        if(!hlpobj.helpon(topic))
          Console.WriteLine("Тема не найдена.\n");

      } while(topic != "stop");
    }
  }
```

Ответы профессионала

Вопрос. Ранее в этой главе уже упоминался класс потоков `MemoryStream` используемый для организации хранилища в памяти компьютера. Каким образов может использоваться этот класс?

Ответ. Класс `MemoryStream` является реализацией класса `Stream`, которая использует байтовый массив для организации ввода/вывода. Ниже приводится вызов одного из конструкторов этого класса.

```
MemorySyteam(byte[] buf)
```

Здесь параметр `buf` представляет собой байтовый массив, используемый в качестве источника или цели для запросов ввода/вывода. Поток, создаваемый с помощью конструктора, можно записывать или считывать. Он поддерживает метод `Seek()`. Необходимо помнить о том, что значение параметра `buf` должно быть достаточно большим, чтобы класс `MemoryStream` мог принять весь адресуемый ему объем данных.

Потоки, использующие хранилища в памяти, достаточно широко используются в программировании. Например, можно организовывать поэтапный комплексный вывод, сохраняя промежуточные фрагменты информации в массиве. Эта техника является особенно полезной при программировании в графической среде, такой как Windows. Можно также перенаправлять стандартный поток для чтения информации из массива. Например, это может оказаться полезным при тестировании программы. И последнее замечание. Для создания символьного потока, основанного на использовании памяти, воспользуйтесь потоками `StringReader` или `StringWriter`.

Контрольные вопросы

1. Почему в C# определяются байтовые и символьные потоки?
2. Какой класс находится на вершине иерархии потоков?
3. Покажите, как следует открывать файл для считывания байтов?
4. Покажите, как следует открывать файл для считывания символов?
5. Для чего служит метод `Seek()`?
6. Какие классы поддерживают двоичный ввод/вывод для встроенных типов данных?
7. Какие методы используются для перенаправления стандартных потоков под управлением программы?
8. Каким образом можно преобразовать числовую строку, такую как «123.23». в ее двоичный эквивалент?
9. Создайте программу, копирующую текстовый файл. В результате выполнения программы все пробелы должны заменяться символами дефисов. Используйте классы файлов байтового потока.
10. Перепишите программу из пункта 9 таким образом, чтобы она использовала классы символьных потоков.

Делегаты, события, пространства имен и дополнительные элементы языка C#

- Делегаты
- События
- Пространства имен
- Создание операторов преобразования
- Препроцессор
- Атрибуты
- Указатели и незащищенный контекст
- Идентификация типа во время работы программы
- Дополнительные ключевые слова в C#

Итак, эта глава является последней в данной книге. В ней мы рассмотрим важные элементы языка C# — делегаты, события и пространства имен, а также расскажем об операторах преобразования, атрибутах, директивах препроцессора и некоторых передовых технологиях, которые применяются главным образом в особых ситуациях.

Делегаты

Термин «делегат» звучит необычно для новичков, изучающих C#. На самом деле делегаты в понимании и использовании не сложнее любых других элементов этого языка. По своей структуре *делегат* — это объект, который может ссылаться на метод. То есть при создании делегата создается объект, который может хранить ссылку на метол. Кроме того, эта ссылка может быть использована для вызова метода. Следовательно, делегат может вызывать метод, на который он ссылается.

Хотя метод не является объектом, для него выделяется некоторая область памяти. Адрес этой области памяти — это точка входа метода, и именно адрес инициируется при вызове метода. Значение адреса метода (области памяти) может быть присвоено делегату. Если делегат ссылается на метод, то данный метод можно вызывать с помощью этого делегата. Кроме того, один и тот же делегат может использоваться при вызове различных методов, для этого ему присваивается ссылка на требуемый метод. Важным свойством делегата является то, что он позволяет указать в коде программы вызов метода, но фактически вызываемый метод определяется во время работы программы, а не во время компилирования.

Примечание

Если вы знакомы с языками C/C++, вам будет интересно узнать, что делегаты в C# сходны с указателями на функции в C/C++.

Для объявления делегата необходимо использовать ключевое слово `delegate`. Общая форма синтаксиса объявления делегата показана ниже,

```
delegate ret-type name (parameter-list);
```

Здесь словосочетание `ret-type` — тип значения, возвращаемого методами, которые будут вызваны делегатом. Слово `name` — имя делегата. Параметры, необходимые методам при их вызове с помощью делегата, указываются в списке параметров `parameter-list`. Делегат может вызывать только те методы, типы возвращаемых значений и списки параметров которых совпадают с типами и списками, указанными при объявлении делегата.

Как уже говорилось, характерной особенностью делегата является возможность его использования для вызова любого метода, который соответствует подписи делегата. Это дает возможность определить во время выполнения программы, какой из методов должен быть вызван. Вызываемый метод может быть методом экземпляра, ассоциированным с объектом, либо статическим методом, ассоциированным с классом. Метод можно вызвать только тогда, когда его подпись соответствует подписи делегата.

Рассмотрим простую программу, в которой используется делегат.

```
// Простая программа, в которой демонстрируется использование делегата.

using System;
```

```
// Объявление делегата.
delegate string strMod(string str);

class DelegateTest {
  // Метод заменяет пробелы дефисами в строке, передаваемой ему в качестве
  // параметра.
  static string replaceSpaces(string a) {
    Console.WriteLine("Замена пробелов дефисами.");
    return a.Replace(' ', '-');
  }

  // Метод, предназначенный для удаления пробелов в передаваемой ему строке.
  static string removeSpaces(string a) {
    string temp = "";
    int i;

    Console.WriteLine("Удаление пробелов.");
    for(i=0; i < a.Length; i++)
      if(a[i] != ' ') temp += a[i];

    return temp;
  }

  // Метод выводит передаваемую ему строку в обратном порядке.
  static string reverse (string a) {
    string temp = "";
    int i, j;

    Console.WriteLine("Вывод строки в обратном порядке.");
    for(j=0, i=a.Length-1; i >= 0; i--, j++)
      temp += a[i];

    return temp;
  }

  public static void Main() {
    // Создание делегата.
    strMod strop = new strMod(replaceSpaces);
    string str;

    // Вызов метода с помощью делегата.
    str = strOp("Проверка работы методов.");
    Console.Writeline("Преобразованная строка: " + str);
    Console.WriteLine();

    strOp = new strMod(temoveSpaces);
    str = strOp("Проверка работы методов.");
    Console.Writeline("Преобразованная строка: " + str);
    Console.WriteLine();

    strOp = new strMod(reverse);
    str = strOp("Проверка работы методов.");
    Console.WriteLine("Преобразованная строка: " + str);
  }
}
```

Делегат strMod.

Создание делегата.

Вызов метода с использованием делегата.

Результат выполнения программы:

```
Замена пробелов дефисами.
Преобразованная строка: Проверка-работы-методов.

Удаление пробелов.
Преобразованная строка: Проверкаработыметодов.

Вывод строки в обратном порядке.
Преобразованная строка: .водотем ытобар акреворП
```

Рассмотрим эту программу более подробно. В ней объявляется делегат `strMod`, который принимает один параметр типа `string` и возвращает значение типа `string`. В классе `DelegateTest` объявляются три метода с модификатором `static` с соответствующей делегату подписью. Эти методы предназначены для преобразования строки, передаваемой им в качестве параметра. Обратите внимание, что метод `replaceSpases()` для замены пробелов дефисами использует один из методов класса string, который называется `Replace()`.

В методе `Main()` создается объект `strOp` типа `strMod`, которому присваивается ссылка на метод `replaceSpaces()`. В операторе

```
strMod strOp = new strMod(replaceSpaces);
```

метод `replaceSpaces()` передается конструктору делегата в качестве параметра. Используется только имя метода без указания параметра (то есть при создании экземпляра делегата указывается только имя метода, на который должен ссылаться делегат). Подпись метода должна соответствовать подписи, определенной в объявлении делегата. Если это условие не выполнено, при компилировании программы произойдет ошибка компиляции.

В следующей строке кода экземпляр делегата `strOp` используется для вызова метода `replaceSpaces()`.

```
sir = strop("Проверка работы методов.");
```

Поскольку объект `strOp` ссылается на метод `replaceSpases()`, то вызывается именно этот метод.

Далее в программе экземпляру делегата `strOp` присваивается ссылка на метод `removeSpaces()`, после чего экземпляр делегата `strOp` вызывается вновь. На этот раз инициируется метод `removeSpaces()`.

В завершение программы экземпляру делегата `strOp` присваивается ссылка на метод `revers()`, после чего вновь выполняется оператор

```
str = strOp("Проверка работы методов.");
```

На этот раз вызывается метод `revers()`. В данной программе при каждом вызове экземпляра делегата `strOp` будет вызываться тот метод, на который ссылается делегат `strOp` во время вызова. Таким образом, вызываемый метод определяется во время выполнения программы, а не во время компилирования.

В этом примере использовались методы с модификатором `static`, но делегатам может присваиваться также ссылка на методы экземпляра. Например, ниже приведена новая версия предыдущей программы, в которой операции со строкой инкапсулированы внутри класса `StringOps`.

```
// Новая версия предыдущей программы, в которой делегаты ссылаются на
// методы экземпляра.

using System;

// Объявление делегата.
delegate string strMod(stnng str);

class StringOps {
   // Метод заменяет пробелы дефисами в строке, передаваемой ему в качестве
   // параметра.
   public string replaceSpaces(string a) {
     Console.WriteLine("Замена пробелов дефисами.");
     return a.Replace(' ', '-');
   }

   // Метод, предназначенный для удаления пробелов в передаваемой ему строке.
   public string removeSpaces(string a) {
     string temp — "";
     int i;

     Console.WriteLine("Удаление пробелов.");
     for(i=0; i < a.Length; i++)
       if(a[i] != ' ') temp +— a[i];

     return temp;
   }

   // Метод выводит передаваемую ему строку в обратном порядке.
   public string reverse(string a) {
     string temp = "";
     int i, j;

     Console.WriteLine("Вывод строки в обратном порядке.");
     for(j=0, i=a.Length-1; i >= 0; i--, j++)
       temp += a[i];

     return temp;
   }
}

class DelegateTest {
   public static void Main() {
     StringOps so = new StringOps();

     // Создание экземпляра делегата.
     strMod strOp = new strMod(so.replaceSpaces);
     string str;

     // Вызов метода с помощью делегата.
     str = strOp("Проверка работы методов.");
     Console.WriteLine("Преобразованная строка: " + str);
     Console.WriteLine();

     strOp = new strMod(so.removeSpaces);
     str = strOp("Проверка работы методов.");
```

Создание делегата с использованием метода экземпляра.

```
    Console.WriteLirve("Преобразованная строка: " + str);
    Console.WriteLine();

    strOp = new strMod(so.reverse);
    str = strOp("Проверка работы методов.");
    Console.WriteLine("Преобразованная строка: " + str);
  }
}
```

В результате работы этой программы на экран выводятся те же строки, что и в предыдущей. Но в данном случае делегат ссылается на методы, используя экземпляр класса `StringOps`.

Многоадресность делегатов

Одним из замечательных свойств делегата является его способность хранить несколько адресов области памяти. Последовательно инициируя эти адреса, делегат может один за другим вызывать соответствующие методы. Эта характеристика делегатов называется *многоадресностью*. Такая последовательность вызываемых методов, или *цепочка методов*, конструируется достаточно просто — сначала необходимо создать экземпляр делегата, а затем использовать оператор += для добавления методов к цепочке. Хотя фактически при этом создается не цепочка методов, а цепочка ссылок на методы, в результате чего этот делегат становится *многоадресным*. Для удаления метода из цепочки используется оператор -=. (Конечно, для добавления и удаления методов из цепочки можно по отдельности использовать операторы +, - и =, но применение составных операторов присваивания +-= и -= более удобно.) Единственное ограничение — делегаты, хранящие несколько ссылок, должны иметь тип возвращаемого значения `void`.

Ниже приведена программа, демонстрирующая использование многоадресного делегата. Это новая версия предыдущей программы, в которой в качестве типа возвращаемого значения методов, преобразующих строку, используется тип `void`, а для возвращения вызывающей подпрограмме измененной строки применяется модификатор параметра `ref`.

```
// В программе демонстрируется использование многоадресности делегата.

using System;

// Объявление делегата.
delegate void strMod(ref string str);

 class StringOps {
   // Метод заменяет пробелы дефисами в строке, передаваемой ему в качестве
   // параметра.
  static void replaceSpaces(ref string a) {
    Console.Writeline("Замена пробелов дефисами.");
    a = a.Replace( ' ', '-');


  // Метод, предназначенный для удаления пробелов в передаваемой ему строке.
  static void removeSpaces(ref string a) {
    string temp = "";
    int i;
    Console.WriteLine("Удаление пробелов.");
    for(i=0; i < a.Length; i++)
```

```
      if(a[i] != ' ') temp += a[i];

    a = cemp;
  }

  // Метод  выводит передаваемую ему строку в обратном порядке.
  static void reverse(ref string a) {
    string temp = "";
    int i, j;

    Console.WriteLine("Вывод строки s обратном порядке.");
    for(j=0, i=a.Length-1; i >= 0; i--, j++)
      temp += a[i];

    a = temp;
  }

  public static void Main() {
    // Создание экземпляра делегата.
    strMod strOp;
    strMod replaceSp = new strMod(replaceSpaces);
    strMod removeSp = new strMod(removeSpaces);
    strMod reverseStr = new strMod(reverse);
    string str = "Проверка работы методов.";

    // Делегату присваиваются две ссылки.
    strOp = replaceSp;
    strOp += reverseStr;

    // Вызов многоадресного делегата.
    strOp(ref str);
    Console.WriteLine("Преобразованная строка: " + str);
    Console.WriteLine();

    // Удаление из цепочки ссылок ссылки на метод replaceSpaces и добавление
    // ссылки на метод removeSpaces
    strOp -= replaceSp;
    strOp += removeSp;

    str = " Проверка работы методов."; // Присвоение строковой переменной str
                  // строки "Проверка работы методов" в первоначальном виде.

    // Вызов многоадресного делегата.
    strOp(ref str);
    Console.WriteLine("Преобразованная строка: " + str);
    Console.WriteLine();
  }
}
```

Создание цепочки ссылок.

Создание новой цепочки ссылок.

Ниже приведен результат выполнения этой программы.

```
Замена пробелов дефисами.
Вывод строки в обратном порядке.
Преобразованная строка: .водотем—ытобар—акреворП

Вывод строки в обратном порядке.
Удаление пробелов.
Преобразованная строка: .водотемытобаракреворП
```

В методе `Main()` создаются четыре экземпляра делегата. Экземпляр `strOp` имеет значение `null`, поскольку при создании он не был инициирован. Остальные три

делегата ссылаются на методы, предназначенные для преобразования строки. Далее в программе создается многоадресный делегат, который затем вызывает методы `removeSpaces()` и `reverse()`. Это осуществляется с помощью трех операторов:

```
strOp = replaceSp;
strOp += reverseStr;
strOp(ref str);
```

В первом операторе экземпляру `strOp` делегата `strMod` присваивается ссылка на метод `replaceSp`. Затем с помощью оператора `+=` к цепочке присоединяется ссылка на метод `reverseStr`. При вызове объекта `strop` вызываются оба метода. Первый метод заменяет пробелы дефисами, а второй изменяет порядок символов в строке.

В операторе

```
strOp -= replaceSp;
```

сначала из цепочки удаляется ссылка объекта `replaceSp`, а затем к цепочке прибавляется ссылка объекта `removeSp`:

```
3trOp += removeSp;
```

Далее в программе вновь вызывается объект `strOp` с еще не преобразованной строкой в качестве аргумента. Теперь из строки удаляются пробелы, и она выводится в обратном порядке.

Преимущества использования делегатов

В предыдущем разделе описывалось использование делегатов, но не объяснялось, для чего был создан этот элемент языка. Делегаты в C# применяются по двум причинам, во-первых, они поддерживают события (которые мы рассмотрим в следующем разделе), во-вторых, они предоставляют возможность выбора вызываемого метода во время выполнения программы, а не во время компилирования. Эта способность весьма полезна, когда необходимо создание базовой конструкции (программы), в которую потом можно добавлять компоненты. Представьте себе программу для создания рисунков (подобную стандартной программе Windows Paint). В нее можно добавлять специальные световые фильтры; или анализаторы изображений. Кроме того, программист может создать последовательность таких фильтров или анализаторов. При использовании делегатов эта схема модификации программы легко реализуется.

Минутный практикум

1. Что такое делегат? Какие преимущества дает использование делегатов?
2. Как объявляется делегат?
3. Что такое многоадресность?

1. Делегат — это объект, имеющий ссылку на метод. Делегат позволяет выбрать вызываемый метод во время выполнения программы.

2. Делегат объявляется с помощью ключевого слова `delegate`, за которым указывается тип возвращаемого значения, имя делегата и список параметров вызываемых методов.

3. Многоадресность — это способность делегата хранить несколько ссылок на различные методы, что позволяет при вызове делегата инициировать эту цепочку методов.

События

Еще одним важным элементом языка C#, построенным на основе использования делегатов, является *событие*. Фактически событие — это автоматическое извещение о каком-либо произошедшем действии. Функционируют оно следующим образом: для объекта, который в соответствии с его определением (кодом) должен реагировать на какое-то событие, регистрируется обработчик этого события. Когда событие происходит, вызываются все зарегистрированные обработчики. Обработчики событий создаются на основе делегатов.

События являются членами класса и объявляются с использованием ключевого слова `event`. Общий синтаксис объявления события представлен ниже.

```
event event-delegate object-name;
```

Здесь словосочетание `event-delegate` — это имя делегата, используемого для поддержки события, а словосочетание `object-name` — имя создаваемого объекта события.

Рассмотрим следующую программу:

```
// Очень простой пример использования события.

using System;

// Объявление делегата, на основе которого будет определено событие.
delegate void MyEventHandler();

// Объявление класса, в котором инициируется событие.
class MyEvent {
  public event MyEventHandler activate;

  // В этом методе инициируется событие,
  public void fire() {
    if(activate != null)
      activate();
  }
}

class EventDemo {
  static void handler() {
    Console.WriteLine("Произошло событие.");
  }

  public static void Main() {
    MyEvent evt = new MyEvent();

    // Добавление метола handler() к цепочке обработчиков события.
    evt.activate += new MyEventHandler(handier);

    // Инициирование события.
    evt.fire();
  }
}
```

Объявление делегата для события.

Объявление события.

Определение условия для инициализации события.

Создание экземпляра события.

Добавление метода к (возможной) цепочке обработчиков события.

Инициирование события.

Результат выполнения этой программы следующий:

```
Событие произошло.
```

Хотя данная программа достаточно проста, в ней содержатся все основные элементы, необходимые для правильной обработки события. Давайте рассмотрим ее подробнее.

Программа начинается с объявления делегата для хранения ссылок на обработчики события.

```
deleg ate void MyEventHandler();
```

Все обработчики события активизируются с помощью делегата. Следовательно, делегат события определяет подпись события. В данном примере событие не имеет параметров, но их наличие допускается. Поскольку, как правило, событие является многоадресным, при его объявлении должен указываться тип возвращаемого значения `void`.

Далее в программе создается класс `MyEvent`, в котором объявляется событие `activate` и определяются условия его инициирования. Событие объявляется в следующей строке кода:

```
public event MyEventHandler activate;
```

Обратите внимание на используемый синтаксис. Так объявляются все типы событий.

Внутри класса `MyEvent` объявляется также метод `fire()`, который будет вызываться программой для инициализации события. Условие, при котором выполняется инициализация, определено с помощью оператора `if`.

```
if(activate != null)
   activate();
```

Обратите внимание, что обработчик вызывается только в случае, если делегат, ассоциированный с событием `activate`, имеет значение, отличное от `null`.

Внутри класса `EventDemo` создается обработчик события `handler()`. В этом примере обработчик события просто выводит сообщение. Но можно определить обработчик гак, чтобы его операторы выполняли более содержательные действия. В методе `Main()` создается объект типа `MyEvent`, а метод `handler()` регистрируется как обработчик для события, объявленного в этом классе.

```
KyEvent evt = new MyEvent();

   // Добавление метода handler() к цепочке обработчиков события.
 evt.activate new MyEventHandler(handler);
```

Обратите внимание, что обработчик добавляется с использованием оператора +=. События поддерживают только применение операторов += и −=.

Наконец, происходит инициирование события.

```
   // Инициирование событие.
 evt.fire();
```

Вызов метода `fire()` приводит к вызову всех зарегистрированных обработчиков события. В данном случае вызывается только один зарегистрированный обработчик, но их может быть и несколько.

Широковещательное событие

Как уже говорилось ранее, события создаются на основе делегатов. А поскольку делегаты могут быть многоадресными, то события могут активизировать несколько обработчиков, даже те. которые были определены п других объектах. Такие события называются *широковещательными* (что является синонимом слова многоадресные). Их использование позволяет множеству объектов «реагировать» на извещение о событии. Ниже приведена программа, в которой используется широковещательное событие.

```
// В программе демонстрируется использование широковещательного события.

using System;

// Объявление делегата, на основе которого будет определено событие.
delegate void MyEventHandler();

// Объявление класса, в котором инициируется событие.
class MyEvent {
  public event MyEventHandler activate;

  // В этом методе инициируется событие.
  public void fire() {
    if (activate != null)
      activate();
  }
}

class X {
  public void Xhandler() {
    Console.Writeline ("Событие получено объектом класса X.");
  }
}

class Y {
  public void Yhandler() {
    Console.WriteLine("Событие получено объектом класса Y.");
  }
}

class EventDemo {
  static void handler() {
    Console.Writeline("Событие получено объектом класса EventDemo.");
  }
}
  public static void Main() {
    MyEvent evt = new MyEvent();
    X xOb = new X();
    Y yOb = new Y();

    // Добавление методов handler(),Xhandler() и Yhandler() в цепочку
    // обработчиков события.
    evt.activate += new MyEventHandler(handler);
    evt.activate += new MyEventHandler(xOb.Xhandler);
    evt.activate += new MyEventHandler(yOb.Yhandler);
```

Создание цепочки обработчиков для события.

```
    // Инициирование события.
    evt.fire();
    Console.WriteLine();

    // Удаление одного обработчика из цепочки.
    evt.activate -= new MyEventHandler(xOb.Xhandler);
    evt.fire();
  }
}
```

Результат выполнения этой программы следующий:

```
Событие получено объектом класса EventDemo.
Событие получено объектом класса X.
Событие получено объектом класса Y.

Событие получено объектом класса Y.
Событие получено объектом класса EventDemo.
```

В этой программе создаются два дополнительных класса `X` и `Y`, в которых также определены обработчики событий, совместимые по своей подписи с типом `MyEventHandler`. Следовательно, эти обработчики могут стать частью цепочки. Обратите понимание, что обработчики, определенные в классах `X` и `Y`, не имеют модификатора `static`. Это означает, что сначала в программе создаются объекты каждого класса, а уже затем добавляется относящийся к объекту обработчик.

Необходимо понимать, что сгенерированные события передаются отдельно каждому экземпляру объекта, а не классу в целом. Следовательно, для получения извещения о событии должен быть зарегистрирован (определен) обработчик каждого объекта класса. Например, в приведенной ниже программе происшедшее событие (вызов метода `fire`) инициирует обработчики трех объектов класса `X`.

```
// На происшедшее событие реагируют обработчики каждого из объектов класса.

using System;

// Объявление делегата, на основе которого будет определено событие.
delegate void MyEventHandler();

// Объявление класса, в котором инициируется событие.
class MyEvent {
  public event MyEventHandler activate;

  // В этом методе инициируется событие.
  public void fire() {
    if(activate != null)
      activate();
  }


class X {
  int id;

  public X(int x) ( id = x; )

  public void Xhandler() {
    Console.WriteLine("Событие получено объектом №" + id);
  }
```

```
}

class EventDemo {
  public static void Main() {
    MyEvent evt = new MyEvent();
    X o1 = new X(1);
    X o2 = new X(2);
    X o3 = new X(3);

    evt.activate += new MyEventHandler(o1.Xhandler);
    evt.activate += new MyEventHandler(o2.Xhandler);
    evt.activate += new MyEventHandler(o3.Xhandler);

    // Инициирование события.
    evt.fire();
  }
}
```

В результате выполнения этой программы будет выведена следующая информация:

```
Событие получено объектом №1
Событие получено объектом №2
Событие получено объектом №3
```

Как видите, для каждого объекта отдельно регистрируется обработчик события, и можно сказать, что каждый объект получает отдельное извещение о происшедшем событии.

Примечание

В C# разрешается создание любого типа события. Однако для совместимости компонентов со средой .NET Framework необходимо придерживаться традиций программирования, свойственных компании Microsoft.

Минутный практикум

1. Что такое событие в C#, и какое ключевое слово используется для его объявления?
2. Чем для события является делегат?
3. Могут ли события быть широковещательными?

Пространства имен

Еще в главе 1 мы говорили, что одним из базовых понятий C# является пространство имен. Фактически, так или иначе, пространство имен используется в каждой C#-программе. До этой главы у нас не было необходимости в деталях рассматривать пространства имен, поскольку его применение обеспечивается в Сопрограммах по умолчанию.

1. Событие — это извещение о каком-либо изменении в программе. Для объявления события используется ключевое слово `event`.

2. Делегат служит основой для создания события.

3. Да.

Сначала вспомним, что нам уже известно о пространстве имен. *Пространство имя* определяет область объявления, что позволяет хранить каждый набор имен (класс интерфейсов) отдельно от других наборов. В C# имена, объявленные в одном пространстве имен, не конфликтуют с такими же именами, объявленными в другом пространстве имен. Библиотекой .NET Framework (библиотекой C#) используется пространство имен `System`, поэтому каждая C#-программа начинается строкой

```
using System;
```

Как вам известно из главы 11, классы, предназначенные для выполнения операции ввода/вывода, определяются в пределах пространства имен, подчиненного (находящегося внутри) пространству имен `System`, которое называется `System.IO`. Кроме этого, существует множество других пространств имен, подчиненных пространству имен `System`, в которых хранятся другие части библиотеки C#.

Пространство имен является очень важной составляющей языка C#, поскольку последние годы было создано огромное количество переменных, методов, свойств классов. В пространство имен могут быть включены библиотечные программы, которые других программистов и организаций, а также ваши собственные коды. Без применения пространства имен попытки использования классов с одинаковыми имена, приводили бы к возникновению конфликтов при компоновке программы (например если вы определите в своей программе класс с именем `Finder`, то возникнет конфликт с другим классом `Finder`, входящим в состав библиотеки, создан! другим программистом и применяемой в вашей программе). К счастью, в C# проблема решена — использование пространства имен позволяет ограничит область видимости имен, объявленных в его пределах.

Объявление пространства имен

Пространство имен объявляется с помощью ключевого слова `namespace`. Обт синтаксис объявления пространства имен показан ниже.

```
namespace name {
    // члены класса
}
```

Здесь *name* — название пространства имен. Все элементы, которые определены внутри пространства имен (это могут быть классы, структуры, делегаты, интерфейсы или другие пространства имен) находятся в пределах области видимости этого пространства имен.

Ниже приведен пример создания пространства имен `Counter`. В нем локализуем имя — класс `CountDown`, используемый для реализации простого счетчика обратного отсчета.

```
   // Объявление пространства имен, локализующего классы, в которых реализованы
   // различные способы подсчета чего-либо.
namespace Counter {
  // Класс, предназначенный для обратного отсчета.
  class CountDown {
    int val;

    public CountDown(int n) { val = n; }

    public void reset (int n) {
```

Объявление пространства имен `Counter`.

```
      val = n;
    }

    public int count() {
      if(val > 0) return val--:
      else return 0;
    }
  }
```

В приведенном выше коде класс `CountDown` объявляется в пределах области видимости, определяемой пространством имен `Counter`.

Теперь рассмотрим программу, в которой демонстрируется использование пространства имен `Counter`.

```
// В программе демонстрируется использование пространства имен.
 using System;

// Объявление пространства имен, локализующего классы,
// в которых реализованы различные способы
// подсчета чего-либо.
namespace Counter {
  // Класс, предназначенный для обратного отсчета.
  class CountDown {
    int val;

    public CountDown(int n) ( val = n; )

    public void reset(int n) {
      val = n;
    }

    public int count() {
      if(val > 0) return val--;
      else return 0;
    }
  }
}

class NSDemo {
  public static void Main() {
    Counter.CountDown cd1 = new Counter.CountDown(10);
    int i;

    do {
      i = cd1.count();
      Console.Write(i + " ");
    } while (i > 0);
    Console.WriteLine();

    Counter.CountDown cd2 = new Counter.CountDown(20);

    do {
      i = cd2.count();
      Console.Write(i + " ");
    } while(x > 0);
```

Здесь пространство имен `Counter` используется для полного определения местонахождения класса `CountDown`.

```
    Console.WriteLine();
    ca2.reset(4);
    do {
      i = ca2.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
  }
}
```

Результат выполнения программы выглядит следующим образом:

```
   10 9 3 7 6 5 4 3 2 1 0
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
4 3 2 1 0
```

Подробно проанализируем некоторые важные моменты этой программы. Поскольку класс `CountDown` объявляется в пределах пространства имен `Counter`, при создании объекта имя класса `CountDown` должно указываться вместе с пространством имен `Counter`, как показано ниже:

```
Counter.CountDown cd1 = new Counter.CountDown(10);
```

Однако после создания объекта нет никакой необходимости указывать пространстве имен перед именем объекта или его членов. Следовательно, метод `cd1.count()` можно вызывать непосредственно:

```
i = cd1.count();
```

Директива `using`

Как уже говорилось в главе 1, очень утомительно каждый раз указывать пространство имен, когда в программе имеют место частые обращения к членам этого пространства имен. Использование директивы `using` облегчит вам эту задачу. Для всех программ данной книги пространство имен `System` делается видимым путем его указания после директивы `using`. Следовательно, директива `using` может также использоваться для того, чтобы сделать видимыми создаваемые вами пространства имен.

Существуют две формы синтаксиса директивы `using`. Первая форма выглядит так:

```
using name;
```

Здесь слово `name` указывает имя пространства имен, к которому необходимо получить доступ. Эту форму синтаксиса директивы `using` вы уже встречали. Все члены, определенные в пределах указанного пространства имен, становятся доступными (то есть становятся частью текущего пространства имен) и могут использоваться без полного определения местонахождения. Директива `using` должна быть указана в начале каждого файла, перед всеми другими объявлениями.

Ниже показана новая версия предыдущей программы, в которой директива `using` используется для того, чтобы сделать создаваемое пространство имен видимым.

```
   // В программе демонстрируется использование пространства имен.
 using System?

   // С помощью директивы using пространство имен Counter становится видимым
// для кода этой программы.
```

```
using Counter;
```

Использование директивы using делает видимым пространство имен Counter.

```
// Объявление пространства имен, локализующего классы, в которых реализованы
// различные способы подсчета чего-либо.
namespace Counter {
  // Класс, предназначенный для обратного отсчета.
  class CountDown {
    int val;

    public CountDown(int n) { val = n; }

    public void reset(int n) {
      val = n;
    }

    public int count()  {
      if(val > 0) return val--;
      else return 0;
    }
  }
}

class NSDemo {
  public static void Main() {
    // Теперь класс CountDown может использоваться непосредственно без
    // указания его местонахождения.
    CountDown cd1 = new CountDown(10);
    int i;

    do {
      i = cd1.count();
      Console.Write(i + " ;
    } while(i > 0);
    Console.WriteLine();

    CountDown cd2 = new CountDown(20);

    do {
      i = cd2.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    cd2.reset(4);
    do {
      i = cd2.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
  }
}
```

Теперь ими (класс) CountDown используется непосредственно.

Обратите внимание на еще один важный момент. В этой программе одно пространство имен не скрывает другого пространства имен. При объявлении пространства

имен видимым оно добавляет свои имена к другим доступным в настоящий момент именам. Таким образом, оба пространства имен, `System` и `Counter`, становятся видимыми для кода этой программы.

Вторая форма синтаксиса директивы `using`

Директива `using` имеет и вторую форму, показанную ниже.

```
using alias = name;
```

Здесь слово `alias` (*псевдоним*) становится другим именем для класса или пространства имен, указываемого вместо слова *name*. Далее представлена еще одна версия предыдущей программы, в которой создается псевдоним `Count` класса `Counter.CountDown`.

```
// В программе демонстрируется использование псевдонима.
using System;

// Создание псевдонима класса Counter.CountDown.
using Count = Counter.CountDown;

// Объявление пространства имен, локализующего классы, в которых реализованы
// различные способы подсчета чего-либо.
 namespace Counter {
  // Класс, предназначенный для обратного отсчета.
  class CountDown {
    int val;

    public CountDown(int n) { val = n; }

    public void reset (int n) {
      val = n;
    }

    public int count() {
      if (val > 0) return val--;
      else return 0;
    }

  }
}

class NSDemo {
  public static void Main() {
    Count cd1 = new Count(10);
    int i;

    do {
      i = cd1.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    Count cd2 = new Count(20);

    do {
      i = cd2.count();
```

Создание псевдонима класса `Counter.CountDown`.

Использование псевдонима.

```
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    cd2.reset(4);
    do {
      i = cd2.count();
      Console.Write(i + " ");
    } while (x > 0);
    Console.WriteLine();
  }
}
```

После создания псевдоним `Count` класса `Counter.CountDown` можно использовать для объявления объектов без указания пространства имен. Например, строка кода

```
Count cd1 = new Count(10);
```

создает объект типа `CountDown`.

Аддитивность пространств имен

В C# существует возможность использования одного имени для объявления более одного пространства имен. Это позволяет в одной программе делать видимыми несколько пространств имен, имеющих одинаковое имя. Например, в следующей программе определены два пространства имен `Counter`. В одном содержится класс `CountDown`, во втором содержится класс `CountUp`. При компилировании члены обоих пространств имен `Counter` становятся видимыми.

```
// В программе демонстрируется аддитивность пространств имен.
using System;

// С помощью директивы using пространства имен Counter становятся видимыми
// для кода этой программы.
using Counter;

// Объявление одного пространства имен Counter.
namespace Counter {
  // Класс, предназначенный для обратного отсчета.
  class CountDown {
    int val;

    public CountDown(int n) { val = n; }

    public void reset (int n) {
      val = n;
    }

    public int count() {
      if (val > 0) return val--;
      else return 0;
    }
  }
}

// Объявление еще одного пространства имен Counter.
namespace Counter {
  // Класс, предназначенный для отсчета в порядке возрастания.
```

В этой программе объявляются два пространства имен Counter.

```
  class CountUp {
    int val;
    int target;

    public int Target { get { return target; } }

    public CountUp (int n) { target - n; val — 0; }

    public void reset(int n) {
      target = n;
      val = 0;
    }

    public int count() {
      if(val < target) return val++;
      else return target;
    }
  }
}

class NSDemo {
  public static void Main() {
    CountDown cd = new CountDown(10);
    CountUp cu = new CountUp(8);
    int i;

    do {
      i = cd.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    do {
      i = cu.count();
      Console.Write(i + " ");
    } while(i < cu.Target);

  }
}
```

При выполнении программы на экран будут выведены числа:

```
10 9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8
```

Обратите внимание, что оператор

```
using Counter;
```

делает видимым все содержимое пространства имен `Counter`. Следовательно, доступ к обоим классам `CountDown` и `CountUp` может осуществляться непосредственно без указания пространства имен. То, что пространство имен `Counter` было разделено на две части, не имеет значения.

Вложенные пространства имен

Пространство имен может быть вложенным. Рассмотрим следующую программу:

```
// В программе демонстрируется использование вложенного пространства имен.
using System;
```

```
namespace NS1 {
  class ClassA {
     public ClassA() {
        Console.WriteLine("Оператор конструктора класса ClassA.");
     }
  }
  namespace NS2 { // Объявление вложенного пространства имен.
    class ClassB {
       public ClassB() {
          Console.WriteLine("Оператор конструктора класса Classes.");
       }
     }
  }
}

class NestedNSDemo {
  public static void Main() {
    NS1.ClassA a = new NS1.ClassA();

 // NS2.ClassB b = new NS2.ClassB (); // Ошибка!!! Местоположение пространства
                                      // имен NS2 не указано полностью,
                                      // поэтому оно не будет видимо для
                                      // кода программы.

    NS1.NS2.ClassB b = new NS1.NS2.ClassB(); // Это правильное полное
                            // определение местонахождения класса ClassB.
  }
}
```

Пространство имен NS2 является вложенным в пространство имен NS1.

Результат выполнения программы будет следующим:

```
Оператор конструктора класса ClassA
Оператор конструктора класса ClassB.
```

В этой программе пространство имен `NS2` является вложенным в пространство имени `NS1`. Поэтому для получения доступа к классу `ClassB` необходимо полностью указать его местоположение, то есть, используя операторы точка (`.`), указать оба пространства имен — пространство имен верхнего уровня `NS1` и вложенное `NS2`.

Вложенное пространство имен можно объявить в одной строке кода, используя одно ключевое слова `namespace` и разделяя пространства имен оператором точка. Например, фрагмент кода

```
namespace OuterNS {
   namespace InnerNS {
      // ...
   }
}
```

можно переписать так:

```
namespace OuterNS.InnerNS {
   // ...
}
```

Пространство имен, используемое по умолчанию

Если в программе не объявляется пространство имен, то используется пространство имен, определенное по умолчанию. Поэтому в предыдущих программах вам не нужно было использовать ключевое слово namespace. Пространство имен по умолчанию удобно применять для коротких простых программ, но код больших прикладных программ должен содержаться в каком-либо пространстве имен. Код необходимо инкапсулировать в пространство имен для предотвращения конфликтов имен.

Пространство имен — это один из элементов языка, помогающий в организации программ и делающий их жизнеспособными в современном сложном сетевом окружении.

Минутный практикум

1. Что определяет пространство имен?
2. Напишите дне формы синтаксиса использования директивы using?
3. Могут ли два пространства имен с одинаковыми именами использоваться в одной программе?

Проект 12-1. Помещение класса Set в пространство имен

Set.cs, SetDemo.cs

В проекте 7-1 был создан класс `Set`, в котором имитировалось множество. Теперь необходимо поместить этот класс в собственное пространство имен, поскольку он может конфликтовать с другими классами, имеющими такое же имя.

Пошаговая инструкция

1. Используя файл `SetDemo.cs` из проекта 7-1, перенесите весь код класса `Set` в файл `Set.cs`. При этом поместите этот код в пространство имен `MyTypes.Set`, как показано ниже:

```
/*
   Проект 12-1

   Помещение класса Set в пространство имен.
*/
using System;

 namespace MyTypes.Set {

  class Set {
    char[] members; // Массив, содержащий символы (имитирующий множество).
    int len; // Количество элементов множества.
```

1. Пространство имен определяет область видимости.
2. `using name` и `using alias = name`.
3. Да.

```
// Создание пустого множества.
public Set() {
   len = 0;
}

// Создание пустого множества заданного размера.
public Set(int size) {
   members = new char[size]; // Выделение памяти для множества.
   len = 0; // Элементы не были сконструированы.


// Конструирование множества на базе другого множества.
public Set(Set s) {
   members = new char[s.len]; // Выделение памяти для множества.
   for(int i=0; i < s.len; i++) members[i] = s[i];
   len = s. len; // Количество элементов множества.
}

// Определение свойства Length, предназначенного только для чтения.
public int Length {
   get {
     return len;
   }
}

// Определение индексатора, предназначенного только для чтения.
public char this[int idx] {
   get {
     if(idx >= 0 & idx < len) return members[idx];
     else return (char)0;
   }
}

/* Если элемент входит в состав множества, метод find возвращает индекс
    элемента, если не входит — возвращается значение -1.
 */
int find(char ch) {
   int i;

   for(i=0; i < len; i++)
     if(members[i] == ch) return i;

   return -1;
}

// Добавление уникального элемента в множество.
public static Set operator +(Set ob, char ch) {
   Sec newset = new Set(ob.len +1); // Увеличение числа элементов
                                                  // нового множества на единицу.

   // Копирование элементов.
   for(int i=0; i < ob.len; i++)
     newset.members[i] = ob.members[i];

   // Переменной len возвращаемого объекта присваивается значение
   // переменной len копируемого объекта, то есть передается размер
```

```
        // множества.
        newset.len = ob.len;

        // Выполняется проверка - имеется ли добавляемый символ в множестве.
        if(ob.find(ch) -- -1) { // Если элемент не найден,
               // он добавляется в новое множество.
          newset.members[newset.len] = ch;
          newset.len++;
        }
        return newset; // Возврат нового множества (объекта класса Set).
      }

      // Удаление элемента из множества.
      public static Set operator -(Set ob, char ch) {
        Set newset = new Set();
        int i = ob.find(ch); // Если элемент не найден,
                             // переменной i присваивается значение 1.

        // Копирование оставшихся элементов с использованием перегруженного
        // оператора +.
        for(int j=0; j < ob.len; j++)
          if(j != i) newset = newset + ob.members[j];

        return newset;
      }

      // Объединение множеств.
      public static Set operator +(Set ob1, Set ob2) {
        Set newset = new Set(ob1); // Копирование первого множества.

        // Добавление уникальных элементов из второго множества.
        for(int i=0; i < ob2.len; i++)
            newset = newset + ob2[i];

        return newset; // Возврат новою множества.


      // Разность множеств.
      public static Set operator -(Set ob1, Set ob2) {
        Set newset = new Set(ob1); // Копирование первого множества.

        // Вычитание из первого множества элементов второго множества.
        for(int i=0; i < ob2.len; i++)
          newset = newset - ob2[i];

        return newset; // Возврат нового множества.
      }
    }
}
```

2. После помещения класса `Set` в пространство имен `MyTypes.Set` для использования этого класса необходимо включить пространство имен `MyTypes.Set` в программу, как это показано ниже:

```
using MyTypes.Set;
```

3. Если этого не сделать, в программе необходимо будет каждый раз при обращении к классу `set` полностью указывать его местоположение, как показано в следующей строке кода:

```
MyTypes.Set.Set s1 = new MyTypes.Set.Set();
```

Операторы преобразования

Иногда возникает необходимость использования объектов класса в выражениях, содержащих другие типы данных. Для решения этой задачи можно использовать перегрузку одного или нескольких операторов. Но чаще всего необходимо простое преобразование из одного типа класса в другой. Для этого в C# предусмотрена возможность создания *операторов преобразования*, с помощью которых объект созданного вами класса можно преобразовать в другой тип.

Существуют две формы синтаксиса операторов преобразования — неявная и явная. Они показаны ниже:

```
public static operator implicit target-type(source-type v) {return value;}
public static operator explicit target-type(source-type v) {return value;}
```

Здесь словосочетание `target-type` — это тип, в который преобразуется объект (конечный тип), словосочетание `source-type` — исходный тип, элемент `v` — ссылочная переменная, которая ссылается на объект, а слово `value` — значение объекта после преобразования. Операторы преобразования возвращают данные только типа `target-type`.

Если при определении оператора указано неявное преобразование `(implicit)`, то преобразование инициируется автоматически, например, при использовании объекта в выражении с переменной, имеющей тип `target-type` (то есть тип, в который и преобразовывался объект при определении оператора преобразования). Если при определении оператора указано явное преобразование `(explicit)`, то преобразование инициируется при использовании приведения типов. Одной паре (объекту исходного типа и типа, в который преобразуется объект) нельзя задавать два оператора преобразования, для одного из которых было бы указано явное преобразование, а для другого неявное.

Чтобы продемонстрировать работу оператора преобразования, используем класс `ThreeD`, созданный в главе 7. Вспомним, что класс `ThreeD` хранит координаты некоторой точки в трехмерном пространстве. Предположим, что необходимо преобразовать объект типа `ThreeD` в целочисленный тип таким образом, чтобы при его использовании в целочисленном выражении содержалось значение, равное произведению значений всех трех координат. Для этого используем оператор неявного преобразования:

```
public static implicit operator int(ThreeD op1)
{
  return op1.x * op1.y * op1.z;
}
```

Далее приведена программа, в которой используется оператор преобразования.

```
// В программе демонстрируется использование оператора преобразования.
using System;
```

```
// Класс, содержащий координаты объекта в трехмерном пространстве.
class ThreeD {
  int x, y, z; // Координаты объекта.

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j , int k) { x = i; y = j; z = k; }

   // Перегрузка бинарного оператора + для пары операндов,
   // имеющих тип ThreeD.
  public static ThreeD operator +(ThreeD op1, ThreeD op2)

    ThreeD result = new ThreeD();

    result.x = op1.x + op2.x;
    result.y = op1.y + op2.y;
    result.z = op1.z + op2.z;

    return result;


  // Определение оператора, для которого указано неявное преобразование из
  // типа ThreeD в тип int.
  public static implicit operator, int(ThreeD op1)

    return op1.x * op1.y * op1.z;
  }

  // Отображение координат X, Y, Z.
  public void show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  public static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD();
    int i;

    Console.Write("Координаты объекта a: ");
    a.show();
    Console.WriteLine();
    Console.Write("Координаты объекта b: ");
    b.show();
    Console.WriteLine();

    c = a + b; // Сложение объектов a и b.
    Console.Write("Результат выполнения операции сложения " +
                  "объектов a и b: ");
    C.show();
    Console.WriteLine();

    i = a; // При выполнении этого оператора присваивания инициируется
           // неявное преобразование объекта a в тип int.
```

Оператор, выполняющий преобразование из типа `ThreeD` в тип `int`.

Инициируется преобразование в тип `int`.

```
    Console.WriteLine("Результат выполнения выражения i = a: " + 1);
    Console.WriteLine();

    i = a * 2 - b; // Выполняется неявное преобразование двух объектов a и b в
                   // тип int.
    Console.WriteLine("Результат выполнения выражения a * 2 - b: " + 1);
  }
}
```

Инициируется преобразование в тип int-

При выполнении программы будут подучены следующие результаты:

```
Координаты объекта a: 1, 2, 3

Координаты объекта b: 10, 10, 10

Результат выполнения операции сложения объектов a и b: 11, 12, 13

Результат выполнения выражения i — a: 6

Результат выполнения выражения a * 2 - b: -988
```

Как показано в программе, при использовании объекта типа `ThreeD` в целочисленном выражении, таком как `i = a`, инициируется оператор неявного преобразования. В данном конкретном случае при выполнении оператора преобразования возвращается значение 6, которое является произведением значений координат, хранящихся в объекте `a`. Однако если выражение не требует преобразования в тип `int`, оператор преобразования не вызывается. Поэтому при выполнении выражения `c — a + b` вызов оператора преобразования `operator int()` не требуется.

Помните, что для решения различных задач можно создавать различные операторы преобразования. Например, можно определить оператор для преобразования в тип `double` или `long`. Каждый такой оператор применяется автоматически и независимо от остальных.

Оператор неявного преобразования применяется автоматически в следующих случаях: когда преобразование требуется в выражении, объект передастся методу, выполняется оператор присваивания, используется явное приведение типов. В качестве альтернативного можно создать оператор явного преобразования, который применяется только при использовании явного приведения типов и не вызывается автоматически. Например, ниже приведена переработанная предыдущая программа, в которой используется явное приведение к типу `int`.

```
// В программе демонстрируется использование оператора явного
// преобразования.
 using System;

// Класс, содержащий координаты объекта в трехмерном пространстве.
class ThreeD {
   int x, y, z; // Координаты объекта.

  public ThreeD() { x = y = z = 0; }
  public ThreeD (int i, int j, int k) { x = i; y = j; z = k; }

    // Перегрузка бинарного оператора + для пары операндов,
   // имеющих тип ThreeD.
  public static ThreeD operator +(ThreeD op1, ThceeD op2)
```

```
  {
    ThreeD result = new ThreeD();

    result.x = op1.x + op2.x;
    result.y = op1.y + op2.y;
    result.z = op1.z + op2.z;

    return result;
   }

  // Теперь определяется оператор явного преобразования.
  public static explicit operator int(ThreeD op1)

    return op1.x * op1.y * op1.z;


  // Отображение координат X, Y, Z.
  public void show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  public static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD();
    int i;

    Console.Write("Координаты объекта a: ");
    a.show();
    Console.WriteLine();
    Console.Write("Координаты объекта b: ");
    b.show();
    Console.WriteLine();

    c = a + b; // Сложение объектов a и b.
    Console.Write("Результат выполнения операции сложения " +
                  "объектов a и b: ");
    c.show();
    Console.WriteLine();

    i = (int) a; // Для выполнения этого оператора присваивания требуется
          // выполнение операции приведения типа, после чего будет вызван
          // оператор явного преобразования объекта a в тип int.
    Console.WriteLine("Результат выполнения выражения i = a:" + i) ;
    Console.WriteLine();

    i = (int)a * 2 - (int)b; // Требуется выполнение операции приведения типа.
    Console.WriteLine("Результат выполнения выражения a * 2 - b: " + i);

  }
```

Теперь преобразование явное.

Теперь требуется выполнение операции приведения типа.

Поскольку теперь оператор преобразования указан с ключевым словом `explicit`, для его инициации и преобразования объекта класса `ThreeD` в тип `int` требуется выполнение операции приведения типа. Например, если в операторе

```
i = (int) a;
```

удалить операцию приведения типа, программа не будет скомпилирована.

Ответы профессионала

Вопрос. Если оператор неявного преобразования вызывается автоматически без использования операции приведения типа, зачем нужно создавать оператор явного преобразования?

Ответ. Хотя неявное преобразование удобно в использовании, вы должны применять его только тогда, когда уверены, что преобразование не приведет к возникновению ошибки. Оператор неявного преобразования можно создавать только в случае выполнения двух условий. Во-первых, при преобразовании не должна быть потеряна информация, что происходит при усечении, переполнении или потере знака. Во-вторых, преобразование не должно приводить к возникновению исключительной ситуации. Если не соблюдается хотя бы одно из этих условий, нужно использовать явное преобразование.

Для использования операторов преобразования существуют некоторые ограничения.

- Нельзя создавать оператор, который выполнял бы преобразование из встроенного типа в какой-либо другой. Например, нельзя переопределить операцию приведения типа `double` к типу `int`.
- В операторе преобразования нельзя использовать тип `object`.
- Нельзя определять неявное и явное преобразование для одной пары типов, состоящей из исходного и конечного типов.
- Нельзя определять преобразование из наследуемого класса в наследующий.
- В операторе преобразования нельзя использовать интерфейс.

Минутный практикум

1. Для чего предназначен оператор преобразования?
2. Чем отличается оператор явного преобразования от оператора неявного преобразования?
3. Можно ли использовать оператор преобразования для преобразования из встроенного типа?

---

1. Оператор преобразования предназначен для преобразования в тип класса или из типа класса.
2. Оператор неявного преобразования вызывается автоматически. Оператор явного преобразования должен вызываться с использованием операции приведения типа.
3. Нет.

Препроцессор

В C# определены несколько *директив препроцессора*, которые влияют на способ интерпретации компилятором исходного файла программы. Эти директивы изменяют текст исходного файла, в котором они встречаются, перед трансляцией программы в объектный код. Директивы препроцессора в основном пришли в C# из C++. Фактически препроцессоры в этих языках очень похожи. Данные инструкции получили название директив препроцессора, поскольку они традиционно обрабатывались в отдельной фазе компилирования, которая называлась *препроцессором.* Современные технологии компилирования больше не требуют отдельной фазы для обработки директив, но имя этих инструкций осталось прежним.

В C# определены следующие директивы препроцессора.

`#acfine`	`#elif`	`#else`	`#endif`
`#endregion`	`#error`	`#if`	`#line`
`#region`	`#undef`	`#warning`	

Вес директивы препроцессора начинаются со знака #. Кроме того, каждая из них должна находиться на отдельной строке программы.

Современная объектно-ориентированная архитектура C# не так нуждается в директивах препроцессора, как архитектура более старых языков, но иногда директивы препроцессора могут быть полезны, особенно для условной компиляции.

Далее по порядку будут рассмотрены все директивы препроцессора.

Директива препроцессора `#define`

С помощью директивы `#define` определяется символьная последовательность, которая называется *символьной константой (symbol).* Наличие или отсутствие конкретной символьной константы определяется с помощью директивы `#if` или `#elif` и может использоваться для управления компилированием. Ниже приведен общий синтаксис директивы `#define`.

```
#define symbol
```

Обратите внимание, что в этом операторе отсутствует точка с запятой. Между директивой `#define` и символьной константой может быть произвольное количество пробелов, а признаком окончания символьной константы является символ новой строки. Например, для определения символьной константы `EXPERIMENTAL` необходимо использовать директиву

```
#define EXPERIMENTAL
```

Ответы профессионала

Вопрос. Я знаю, что в C++ директиву `#define` можно использовать для выполнения текстовых подстановок, например, определения имени для значения или Для создания макросов, аналогичных функциям. Поддерживаются ли в C# такие свойства директивы `#define`?

Ответ. Нет. В C# директива `#define` используется только для определения символьной константы.

Директивы препроцессора #if и #endif

Использование директив препроцессора #if и #endif позволяет условно компилировать последовательность кода. Их выполнение зависит от результата оценки выражения, включающего одну или более символьных констант. Выражение может принимать значение true или faise. Символьная константа принимает значение true, если она определена директивой #define.

Общий синтаксис директивы #if следующий:

```
#if symbol-expression
   statement sequence
#endif
```

Если выражение, следующее за директивой #if, принимает значение true, то код, находящийся между директивами #if и #endif, будет скомпилирован. В противном случае этот код пропускается компилятором. Наличие директивы #endif указывает на окончание блока операторов директивы #if.

Символьное выражение может быть простым и состоять из одной символьной константы. В символьном выражении могут использоваться операторы !, ==, !=, && и ||. Также разрешается применение круглых скобок.

Рассмотрим небольшую программу.

```
// В программе демонстрируемся использование директив препроцессора.
// #if, #endif и #define.
#define EXPERIMENTAL

using System;

class Test {
  public static void Main() {

    #if EXPERIMENTAL
      Console.WriteLine("Этот оператор будет скомпилирован только при" +
  "\nналичии в программе определенной символьной константы EXPERIMENTAL.");
    #endif

    Console.WriteLine("Этот оператор будет скомпилирован независимо от " +
                      "определения символьной константы.");

  }
}
```

Этот оператор будет скомпилирован, поскольку символьная константа EXPERIMENTAL определена директивой #define.

В результате выполнения программы будут выведены следующие строки:

```
Этот оператор будет скомпилирован только при
наличии в программе определенной символьной константы EXPERIMENTAL.
Этот оператор будет скомпилирован независимо от определения символьной
константы.
```

В этой программе определена символьная константа EXPERIMENTAL. Следовательно, когда в коде программы встречается директива #if, символьное выражение принимает значение true и компилируется первый оператор WriteLine();. Если удалить определение символьной константы EXPERIMENTAL и повторно скомпилировать программу, то первый оператор WriteLine(); скомпилирован не будет, поскольку выражение после директивы #if примет значение false. Второй оператор

`WriteLine();` будет скомпилирован и любом случае, поскольку он не является частью блока `#if`.

Как уже говорилось, в блоке `#if` можно использовать символьное выражение. Например, в следующей программе условием для компиляции одного из операторов является наличие двух определенных символьных констант.

```
   // В программе демонстрируемся использование символьного выражения.
#define EXPERIMENTAL
#define TRIAL

  using System;

class Test: {
  public static void Main() {

    #if EXPERIMENTAL
      Console.WriteLine("Оператор будет скомпилирован в той версии " +
 "программы, в которой будет определена символьная константа EXPERIMENTAL.");
    #endif

    #if EXPERIMENTAL && TRIAL
       Console.Error.WriteLine("Оператор Судет скомпилирован в той версии " +
 "программы, в которой будут определены две символьные константы " +
 "EXPERIMENTAL и TRIAL.");
    #endif

    Console.WriteLine("Этот оператор будет скомпилирован независимо от " +
                      "определения символьных констант.");
  }
}
```

Символьное выражение.

Результат выполнения программы следующий:

```
Оператор будет скомпилирован в той версии программы, которой будет
определена символьная константа EXPERIMENTAL.
Оператор будет скомпилирован в той версии программы, в которой будут
определены две символьные константы  и .
  Этот оператор будет скомпилирован независимо от определения символьных
константа.
```

В программе определены две символьные константы — `EXPERIMENTAL` и `TRIAL`. Второй оператор `WriteLine();` будет скомпилирован только в случае определения обеих символьных констант.

Директивы препроцессора `#else` и `#elif`

В C# директива #else работает так же, как оператор else. С се помощью указывается альтернативный вариант блока операторов, который будет скомпилирован, если результатом оценки выражения директивы #if будет значение false. Предыдущая программа может быть переписана следующим образом:

```
   // В программе демонстрируется использование директивы препроцессора #else.

#define EXPERIMENTAL

using System;
```

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

    #if EXPERIMENTAL
      Console.WriteLine("оператор Судет скомпилирован в той версии " +
 "программы, в которой будет определена символьная константа EXPERIMENTAL.");
    #else
      Console.WriteLine("Альтернативный вариант оператора.");
    #endif

    #if EXPERIMENTAL && TRIAL
       Console.Error.WriteLine("Оператор будет скомпилирован в той версии " +
 "программы, в которой будут определены две символьные константы " +
 "EXPERIMENTAL и TRIAL.");
    #else
       Console.Error.WriteLinei"ыде один альтернативный вариант оператора.");
    #endif

    Console.WriteLine("Этот оператор будет скомпилирован независимо от " +
                      "определения символьных констант.");
  }
}
```

Использование директивы #else.

Использование директивы #else.

В этой программе из пяти операторов `Console.WriteLine();` компилируются только три. В результате выполнения этих операторов на экран выводятся следующие строки:

```
Оператор будет скомпилирован в той версии программы, в которой будет
определена символьная константа EXPERIMENTAL.
Еще один альтернативный вариант оператора.
Этот оператор будет скомпилирован независимо от определения символьных констант.
```

Поскольку в этой версии программы символьная константа `TRIAL` не определена, компилятор будет использовать код, указанный для второй директивы `#else`.

Обратите внимание, что наличие в коде программы директивы `#else` означает как конец блока операторов, относящихся к директиве `#if`, так и начало блока операторов, относящихся к директиве `#else`. Это необходимо, поскольку с директивой `#if` может ассоциироваться только одна директива `#endif`.

Предназначение директивы `#elif` аналогично предназначению оператора `else if`. Она используется для создания цепочки `if-else-if`, с помощью которой можно определить несколько вариантов компиляции программы. За директивой `#elif` следует символьное выражение. Если выражение принимает значение true, то компилируется блок кода, идущий за символьным выражением, а выражения, указанные за другими директивами `#elif`, не проверяются. Если выражение принимает значение `false`, то проверяется следующее условное выражение в цепочке. Общая форма синтаксиса директивы `#elif` выглядит так:

```
#if symbol-express!on
   statement sequence
#elif symbol—expression
   statement sequence
#elif symbol—expression
   statement sequence
#elif symbol-expression
   statement sequence
```

```
#elif symbol-expression
   statement sequence
.
.
.
#endif
```

Например,

```
// В программе демонстрируется использование директивы препроцессора #elif.
#define RELEASE

using System;

class Test {
  public static void Main() {

    #if EXPERIMENTAL
      Console.WriteLine("Оператор будет скомпилирован в той версии " +
 "программы, в которой будет определена символьная константа EXPERIMENTAL.");
    #elif RELEASE
      Console.WriteLine("Оператор будет скомпилирован в той версии " +
 "программы, в которой будет определена символьная константа RELEASE.");
    #else
      Console.WriteLine("Оператор будет скомпилирован в той версии " +
 "программы, в которой не будет определена ни одна из перечисленных " +
                        "выше констант. ");
    #endif

    #if TRIAL && !RELEASE
       Console.WriteLine("Оператор будет скомпилирован в той версии " +
 "программы, в которой будет определена символьная константа TRIAL.");
    #endif

    Console.WriteLine("Этот оператор будет скомпилирован независимо от " +
                      "определения символьных констант.");
  }
}
```

Использование директивы #elif.

Программа выводит на экран следующие строки:

```
  Оператор будет скомпилирован в той версии программы, в которой будет
определена символьная константа RELEASE.
Этот оператор будет скомпилирован независимо от определения символьных констант.
```

Директива препроцессора #undef

С помощью директивы препроцессора `#undef` удаляется данное ранее определение символьной константы, которая указывается за директивой. То есть далее в программе указанная символьная константа не будет считаться определенной. Общий синтаксис директивы `#undef` следующий:

```
#undef symbol
```

Например, в таком фрагменте кода:

```
#define SMALL

#if SMALL

#undef SMALL
// С этого места программы символьная константа SMALL не считается
// определенной.
```

После указания в программе директивы #undef символьная константа `SMALL` не считается определенной.

Директива `#undef` используется в основном для локализации символьной константы в тех частях кода, где она необходима.

Директива препроцессора `#error`

С помощью директивы препроцессора #error компилятор получает команду остановить компилирование. Эта директива используется для отладки программы. Ее общий синтаксис выглядит так:

```
#error error-message
```

Если в коде программы встречается директива `#error`, на экран выводится сообщение об ошибке. Например, если компилятор обнаружит в коде строку

```
#error Далее в программе возникает ошибка!
```

компилирование останавливается, и на экран выводится сообщение `Далее в программе возникает ошибка!`.

Директива препроцессора `#warning`

Директива `#warning` аналогична директиве `#error` за исключением того, что при ее использовании предупреждающее сообщение выводится, но компилирование не останавливается. Общий синтаксис директивы `#warning` следующий:

```
#warning warning-message
```

Директива препроцессора `#line`

Директива препроцессора `#line` используется для указания номера строки, с которой будут нумероваться строки следующего за директивой кода, и имени файла, в который будут направляться все сообщения, возникающие при компиляции данного кода. Общий синтаксис директивы `#line` следующий:

```
#line number "filename"
```

Здесь элемент `number` — любое целое число, которое становится номером следующей за директивой строки, а необязательная строка `filename` — это любой действительный идентификатор файла, в который будут направляться сообщения, возникающие

при компиляции. Директива `#line` в основном используется для отладки программы и в специальных приложениях.

Директивы препроцессора `#region` и `#endregion`

Используя директивы `#region` и `#endregion`, можно определить фрагмент кода который будет свернут или вновь развернут с помощью средств интегрированной среды разработки Visual C++ IDE. Общий синтаксис использования этих директив представлен ниже:

```
#region
   // code sequence
#endregion
```

Минутный практикум

1. Какая директива препроцессора используется для определения символьной константы?
2. Для чего применяется директива препроцессора `#if`?
3. Какие операторы разрешено использовать в символьных выражениях, ассоциированных с директивой `#if` или `#elif`?

Атрибуты

В C# есть возможность добавлять в программу описательную информацию в форме *атрибута*. С помощью атрибута определяется дополнительная информация, которая ассоциируется с классом, структурой, методом и так далее. Например, можно создать атрибут, в котором указывается тип кнопки, отображаемой классом.

Атрибуты определяются в квадратных скобках перед элементом, для которого они создаются. Вы можете определить собственные атрибуты либо использовать атрибуты. определенные в C#. Создание собственных атрибутов в этой книге не рассматривается, а встроенные в C# атрибуты `Conditional` и `Obsolete` вы можете успешно использовать.

Атрибут `Conditional`

Атрибут `Conditional` является, возможно, одним из наиболее часто используемых в C#. Он позволяет создавать условные методы. Условный метод выполняется только в том случае, если указанная в атрибуте символьная константа определена с помощью директивы `#define`. В противном случае метод выполняться не будет. То есть Использование условного метода является альтернативой условному компилированию

1. Для определения символьной константы используется директива препроцессора `#define`.
2. Директива препроцессора `#if` применяется для определения блок кода, который будет скомпилирован, если символьное выражение, ассоциированное с этой директивой, принимает значение true. В завершении этого блока кода (операторов) указывается директива #endif.
3. В символьных выражениях, ассоциированных с директивой `#if` или `#elif`, разрешено использование следующих операторов: `!=`, `==`, `&&`, `||` и `!`

с помощью директивы #if. Для использования атрибута Conditional необходимо включить в программу пространство имен System.Diagnostics.

Рассмотрим следующий пример:

```
// В программе демонстрируется использование условного атрибута Conditional.
#define TRIAL

using System;
using System.Diagnostics;

class Test {

  [Conditional("TRIAL")]
  void trial()   {
    Console.WriteLine("Выполнение этого метода зависит от определения символьной" +
                      " константы TRIAL.");
  }

  [Conditional("RELEASE")]
  void release() {
    Console.WriteLine("Выполнение этого метода зависит от определения символьной" +
                      " константы RELEASE.");
  }

  public static void Main() {
    Test t = new Test();

    t.trial(); // Метод будет выполнен только в том случае, если в
               // программе была определена символьная константа TRIAL.
    t.release(); // Метод будет выполнен только в том случае, если в
                 // программе была определена символьная константа RELEASE.
  }
}
```

Указание в условном атрибуте символьной константы TRIAL.

Указание в условном атрибуте символьной константы RELEASE.

При выполнении программы будет вызван метод, выводящий следующую строку:

```
Вызов этого метода зависит от определения символьной константы TRIAL.
```

Внимательно рассмотрим эту программу, чтобы понять, почему в результате выводится именно эта строка. Прежде всего, отметим, что в программе определена символьная константа TRIAL. Далее обратим внимание на то, как кодируются методы trial() и release(). Для обоих методов указан атрибут Conditional, имеющий такой синтаксис:

```
[Conditional symbol]
```

Здесь элемент symbol — это символьная константа, от определения которой зависит, будет ли выполняться данный метод. Этот атрибут может использоваться только с методами. Если символьная константа определена, то при вызове метода его код будет выполняться. Если символьная константа не определена, то метод выполняться не будет.

Внутри метода Main() вызываются оба метода, trial() и release(). Но в программе определена только символьная константа TRIAL, следовательно, выполняется только метод trial(). Вызов метода release() игнорируется. Если определить символьную константу release, то метод release() тоже будет выполняться. Если

определение символьной константы `trial` будет удалено, метод `trial()` выполняться не будет.

Условные методы имеют некоторые ограничения.

- Для них должен быть указан тип возвращаемого значения `void`.
- Они должны быть членами класса, а не интерфейса.
- Перед ними не может стоять ключевое слово `override`.

Атрибут `Obsolete`

Атрибут `System.Obsolete` позволяет отмечать элемент программы как устаревший. Этот атрибут имеет следующий синтаксис:

```
[Obsolete "message"]
```

Здесь словом *message* обозначается сообщение, которое будет выведено на экран при компилировании данного элемента программы. Ниже приведена небольшая программа, в которой используется атрибут `System.Obsolete`.

```
  // В программе демонстрируется использование атрибута Obsolete.
using System;

class Test {

  [Obsolete("Вместо этого метода следует использовать метод myMeth2.")]
  static int myMeth(int a, int b) {
    return a / b;
  }

  // Усовершенствованная версия метода myMeth.
  static int myMeth2(int a, int b) {
    return b == 0 ? 0 : a / b;
  }

  public static void Main() {
    // При компилировании этого оператора будет выведено сообщение.
    Console.WriteLine("4 / 3 = " + Test.myMeth(4, 3));

    // При компилировании этого оператора сообщение, указанное в атрибуте,
    // выведено не будет.
    Console.WriteLine("4 / 3 = " + Test.myMeth2(4, 3));
  }
}
```

В атрибуте указывается строковый литерал, который будет выведен при компилировании метода `myMeth()`.

Если при компилировании программы и методе `Main()` встретится вызов метода `myMeth()`, будет выведено сообщение, рекомендующее программисту использовать метод `myMeth2()`.

Минутный практикум

1. Что такое атрибут?
2. Что такое условный метод?
3. Какие функции выполняет атрибут `Obsolete`?

Небезопасный код

C# позволяет писать так называемый *небезопасный код*. Небезопасным является не тот код, который плохо написан, а тот, выполнение которого не управляется CLR. Как уже говорилось в главе 1, язык *C#* обычно используется для создания управляемого кода. Однако существует возможность написать код, выполнение которого не контролируется CLR. Поскольку неуправляемый код не подвергается такому контролю и ограничениям, как управляемый код, он называется небезопасным. При его использовании нельзя быть уверенным, что он не произведет какие-либо вредные действия. Следовательно, термин *небезопасный* не означает, что код некорректный по существу, имеется в виду лишь то, что этот код потенциально может выполнять действия, не контролируемые CLR.

Конечно, применение управляемого кода предпочтительнее, но он не позволяет использовать *указатели*. Если вы знакомы с языком C или C++, то знаете, что указатели — это переменные, хранящие адреса других объектов. Следовательно, указатели имеют некоторое сходство со ссылками в C#. Основное различие между ними заключается в том, что указатель может содержать любой адрес памяти, а ссылка всегда содержит адрес памяти, выделенной для объекта ее типа. Поскольку указатель может ссылаться на любой адрес памяти, существует возможность его некорректного использования. Кроме того, при использовании указателей можно допустить ошибку в коде. Поэтому в C# не поддерживается использование указателей при создании управляемого кода. Однако они полезны и даже необходимы для некоторых типов программ (например, для утилит системного уровня). Все операции с указателями должны быть отмечены как небезопасные, поскольку они выполняются за пределами управляемой среды.

В общем, если вам необходимо создать код, который выполняется за пределами CLR, лучше использовать язык C++. В C# указатели объявляются и применяются так же, как и в языках C/C++. Но основная цель C# — создание управляемого кода, а способность этого языка поддерживать неуправляемый код используется для решения особых задач. Чтобы скомпилировать неуправляемый код, необходимо применить опцию компилятора `/unsafe`.

Работа с небезопасным кодом — это отдельная сложная тема, которая в данной книге рассматриваться не будет. Но все же мы коротко расскажем об использовании указателей и двух ключевых слов, `unsafe` и `fixed`, применяющихся для поддержки неуправляемого кода.

---

1. Атрибут — это дополнительная информация, ассоциированная с каким-либо элементом (классом, структурой, методом).

2. Условным называется метод, для которого определен атрибут `Conditional`. Условный метод выполняется только в том случае, если была задана указанная в атрибуте символьная константа.

3. Атрибут `Obsolete` инициирует при компиляции вывод предупреждающего сообщения об использовании устаревшего элемента (чаще всего это касается метода).

Краткая информация об указателях

Указатели — это переменные, которые хранят адреса других объектов. Например, если переменная x содержит адрес объекта y, то говорят, что переменная x «указывает» на объект y. Общий синтаксис объявления указателя следующий:

```
type* var-name
```

Здесь слово type — это тип указателя, который не должен быть типом класса. Словосочетание var-name — это имя указателя. Например, для объявления переменной p как указателя на объект типа int используется оператор

```
int* ip;
```

Для указателя с типом float, используется оператор

```
float* fp;
```

В целом, если в операторе объявления перед именем переменной стоит знак *, такая переменная становится указателем.

Тип данных, на который ссылается указатель, определяется его типом. Следовательно, в предыдущих примерах указатель ip может использоваться для ссылки на объект типа int, а указатель fp — для ссылки на объект типа float. Именно потому, что указатели могут содержать адрес любой области памяти, они являются потенциально небезопасными.

Существуют два оператора для работы с указателями — * и &. Оператор & является унарным оператором, который возвращает адрес области памяти, выделенной для его операнда. (Вспомним, что для унарного оператора требуется только один операнд.) Например, в результате выполнения фрагмента кода

```
int* ip;
int num = 10;

ip = &num;
```

переменной ip будет присвоен адрес области памяти, выделенной для переменной num. Этот адрес указывает местоположение переменной в оперативной памяти компьютера, но он не имеет ничего общего со значением переменной num. Следовательно, указатель ip не хранит значение 10 (начальное значение переменной num), он только содержит адрес, по которому хранится эта переменная. Можно сказать, что оператор & возвращает адрес переменной, перед именем которой он используется. Таким образом, оператор ip = # можно прочитать так: «Переменная ip получает адрес переменной num.

Оператор * является дополнением к оператору &. Это унарный оператор, возвращающий значение переменной, адрес которой содержится в его операнде. Например, если переменная ip хранит адрес переменной num, то во фрагменте кода

```
int: val;
val = *ip;
```

переменной val будет присвоено значение 10, то есть значение переменной num, на которую ссылается указатель ip. Можно сказать, что оператор * возвращает значение, хранящееся по адресу, содержащемуся в операнде. В этом случае оператор val = *ip; можно прочесть следующим образом: «Переменная val принимает значение, хранящееся по адресу, который содержится в переменной ip».

Указатели могут также использоваться со структурами. При обращении к члену структуры с помощью указателя нужно использовать оператор -> вместо оператора точка (.). Например, определим простую структуру:

```
struct MyStruct {
   public int x;
   public int y;
   public int sumO { return x + y; );
}
```

Доступ к членам структуры с использованием указателей должен выполняться следующим образом:

```
MyStruct o = new MyStruct();
MyStruct* p; // Объявление указателя.

p = &o;
p->x = 10;
p->y = 20;

Console.WriteLine("Сумма равна: " + p->sum());
```

Ответы профессионала

Вопрос. При объявлении указателя в C++ действие оператора * не распространяется на весь список переменных в операторе объявления. Следовательно, **в** операторе

```
int* p, q;
```

объявляется указатель p с типом int и целочисленная переменная q. То же самое происходит и в следующих двух операторах, в которых объявляются переменные p и q:

```
int* p;
int q;
```

Справедливо ли данное правило для C#?

Ответ. Нет. В C# действие оператора * распространяется на весь список переменных, а оператор

```
int* p; q;
```

создает две переменные, которые становятся указателями. Следовательно, этот оператор равносилен двум операторам

```
int* p;
int* q;
```

Это важное отличие, которое необходимо учитывать при переносе кода C++ в C#.

Ключевое слово `unsafe`

Любой код, в котором используются указатели, должен быть отмечен как небезопасный с помощью ключевого слова unsafe. Отмечать можно как отдельный оператор, так и весь метод. Например, ниже приведена программа, в которой используется указатель внутри метода Main(), полностью объявленного небезопасным. (Для

компилирования программы с небезопасным кодом нужно в командной строке ввести следующую команду: `csc /unsafe filename`, где *filename* — имя компилируемого файла.)

```
   // В программе демонстрируется использование указателя и ключевого слова
   // unsafe.
      using System;

class UnsafeCode {
  // Объявление метола Main как небезопасного.
  unsafe public static void Main() {
    int count = 99;
    int *p;  // Создание указателя, имеющего тип int.

    p = &count; //Указателю p присваивается адрес переменной count.

    Console.WriteLine("Первоначальное значение переменной count = " + *p);

    *p = 10;  // Переменной count присваивается значение с помощью указателя p.

    Console.WriteLine("Новое значение переменной count — " + *p);
  }
}
```

Метод `Main()` отмечен как небезопасный, поскольку в нем используется указатель.

Результат выполнения программы будет следующим:

```
Первоначальное значение переменной count = 99
Новое значение переменной count = 10
```

Ключевое слово `fixed`

Использование модификатора `fixed` предотвращает возможность перемещения управляемого объекта при «сборке мусора» (естественно, сам объект не перемещается, просто при оптимизации памяти для него выделяется другая область, имеющая другой адрес). Этот модификатор необходимо указывать, если на такой объект ссылается указатель. Поскольку указателю ничего не известно о действиях системы «сборки мусора», в случае перемещения объекта он будет ссылаться уже на другой объект. Ниже приведен общий синтаксис использования ключевого слова `fixed`.

```
fixed (type*p = &fixedObj) {
   // Использование фиксированного объекта, для которого
   // указан модификатор fixed.
```

Здесь переменная `p` является указателем, которому присваивается адрес объекта. Объект будет сохраняться в области памяти, выделенной для него на время выполнения блока кода. Модификатор `fixed` можно указывать и для одиночного оператора. но в любом случае он может использоваться только в небезопасном коде.

Рассмотрим пример использования модификатора `fixed`.

```
   // В программе демонстрируется использование небезопасного кода.
using System;

class Test {
  public int num;
  public Test(int i) { num = i; }
```

```
}

class unsafeCode {
  // Объявление метола Main как небезопасного,
  unsafe public static void Main()      {
    Test o *= new Test (19);

    fixed (int *p — &o.nuirT5 {  //  Использование  модификатора  fixed для
                              // присвоения указателю р адреса переменной o.num.

      Console.WriteLine("Первоначальное значение переменной o.num = " + *p) ;

      *p — 10;  // Переменной o.num присваивается значение с помощью
                // указателя р.

      Console.WriteLine("Новое значение переменной o.num " + *p);
    )
  }
}
```

Использование модификатора `fixed` для фиксирования местоположения объекта `o` в памяти.

Результат выполнения этой программы следующий:

```
Первоначальное значение переменной o.num -* 19
Новое значение переменной o.num = 10
```

В программе с помощью модификатора fixed предотвращается перемещение й памяти объекта о. Поскольку указатель р ссылается на переменную о. тип, при. перемещении объекта о адрес в указателе будет неверным.

Мы кратко описали создание и использование небезопасного кода. Если в своих программах вы собираетесь создавать небезопасный код, то должны изучить все эти вопросы более подробно.

Минутный практикум

1. Что такое указатель?
2. Когда используется ключевое слово `unsafe`?
3. Какие функции выполняет модификатор `fixed`?

Идентификация типа во время работы программы

В C# существует возможность определить тип объекта во время выполнения программы. Фактически в C# есть три ключевых слова для поддержки идентификаций типа во время выполнения программы, `is`, `as` и `typeof`. Хотя эти ключевые слова используются только при создании высокопрофессиональных программ, важно иметь о них общее представление.

1. Указатель — это переменная, в которой хранится адрес области памяти, выделенной для другого объекта.

2. Любой код, в котором используются указатели, должен быть отмечен как небезопасный с помощью ключевого слова `unsafe`.

3. Использование модификатора `fixed` предотвращает перемещение объекта во время «сборки мусора».

Проверка типа с помощью ключевого слова is

Принадлежность объекта к определенному типу можно проверить с помощью оператора `is`. синтаксис которого выглядит следующим образом:

```
expr is type
```

Здесь элемент `expr` — это выражение, тип которого сравнивается с типом, указанным справа от оператора `is`. Если эти типы совпадают или совместимы, то результат операции принимает значение `true`. В противном случае — значение `false`. Ниже приведена программа, в которой используется оператор `is`.

```
// В программе демонстрируется использование оператора is.
using System;

class A { }
class B : A {}

class Usels {
  public static void Main() {
    A a = new A();
    B b = new B();

    if(a is A) Console.WriteLine("Объект a имеет тип A.");
    if(b is A)
      Console.WriteLine("Объект b имеет тип A, \n" +
       " поскольку класс B является наследующим по отношению к классу A.");
    if(a is B)
      Console.WriteLine("Этот оператор не будет выполнен, поскольку класс A" +
                        " не наследует класс B.");

    if(b is B) Console.WriteLine("Объект в имеет тип B."};
    if(a is object) Console.WriteLine("Объект a имеет тип Object.");
  }
}
```

Объект `a` не является объектом типа `B`, следовательно, результат выполнения данного оператора `is` принимает значение `false`.

Результат этой операции принимает значение `true`, поскольку объект `a` имеет тип `A`.

В результате работы программы будет выведена следующая информация:

```
Объект a имеет тип A.
Объект b имеет тип A,
Поскольку класс B является наследующим по отношению к классу A.
Объект в имеет тип B.
Объект a имеет тип Object.
```

Большинство выражений с ключевым словом `is` не нуждаются в пояснении, но два из них рассмотрим подробнее. Во-первых, обратите внимание на оператор

```
if(b is A)
  Console.WriteLine("Объект b имеет тип A,\n" +
       " поскольку класс B является наследующим по отношению к классу A.");
```

Выражение `(b is A)` принимает значение `true`, поскольку объект `b` является объектом типа `B`, который наследует класс `A`. Следовательно, тип объекта `b` совместим с типом `A`. Однако класс `A` не наследует класс `B`, следовательно, при выполнении оператора

```
if(a is B)
  Console.WriteLine("Этот оператор не будет выполнен, поскольку класс A" +
                    " не наследует класс B.");
```

условное выражение оператора `if` принимает значение `false`, поскольку объект `a` имеет тип `A`, который не наследует класс `B`. Следовательно, эти типы несовместимы.

Ключевое слово as

В C# существует возможность выполнить операцию приведения типа во время выполнения программы, причем, если преобразование невозможно, это не приведет к возникновению исключительной ситуации. Для выполнения можно использовать оператор `as`, применение которого имеет следующий общий синтаксис:

```
expr as type
```

Здесь элемент `expr` — это выражение, которое приводится к типу, указанному справа от оператора `as`. Если операция приведения типа прошла успешно, возвращается ссылка на тип. В противном случае возвращается пустая ссылка.

Ключевое слово typeof

Указав оператору `typeof` в качестве параметра тип некоторого объекта (или обычный тип), можно получить объект класса `System.Type`. Этот объект будет содержать информацию, ассоциированную с указываемым типом. Оператор имеет следующую общую форму синтаксиса:

```
typeof(type);
```

Здесь элемент `type` — это тип, о котором нужно получить полную информацию. Класс `System.Type` предназначен для получения полной информации о каком-либо типе. Например, при выполнении приведенной ниже программы выводится полное имя класса `StreamReader`:

```
// В программе демонстрируется использование оператора typeof.
 using System;
using System.IO;

class UseTypeof {
  public static void Main() {
    Type t = typeof(StreamReader);
    Console.WriteLine(t.FullName);
  }
}
```

Результат выполнения программы показан ниже.

```
System.IO.StreamReader
```

Минутный практикум

1. При помощи какого оператора можно определить, совместим ли тип объекта с указываемым типом?
2. В чем различие между оператором `as` и операцией приведения типов?
3. Что оператор `typeof` получает в качестве параметра?

---

1. Совместимость типа объекта с указываемым типом можно определить при помощи оператора `is`.

2. Неудачное выполнение операции приведения типа приводит к возникновению исключительной ситуации. При неудачном выполнении оператора `as` возвращается значение `null`.

3. Оператор `typeof` получает, в качестве параметра тип объекта или обычной переменной, о котором нужно получить полную информацию.

Другие ключевые слова

Завершая книгу, мы дадим краткое описание некоторых ключевых слов, о которых до этого не рассказывали.

Модификатор доступа `internal`

В дополнение к модификаторам `public`, `private` и `protected`, которые использовались в программах данной книги и с помощью которых указывалась возможности доступа к членам класса, в C# также определен модификатор `internal`. Этот модификатор объявляет, что член класса известен всем файлам в пределах сборки, но неизвестен за пределами сборки. Сборка — это файл (или файлы), в котором содержатся компоненты программы и информация о ее версии. Следовательно можно сказать, что член класса, указанный как `internal`, становится известным всей программе, но нигде больше.

Ключевое слово `sizeof`

Иногда может потребоваться информация о размере в байтах (количестве байтов выделенных для представления данных) одного из обычных типов C#. Для получении такой информации используется оператор `sizeof`. Он имеет следующий синтаксис:

```
sizeof(type)
```

Здесь элемент `type` — это тип, размер которого определяется. Оператор `sizeof` может использоваться только в небезопасном коде. Следовательно, он предназначен для применения в особых ситуациях, в частности при одновременной работе управляемым и неуправляемым кодом.

Ключевое слово `lock`

Ключевое слово `lock` используется при работе с *множеством потоков.* В C# программа может иметь два и более *потоков выполнения.* Когда это происходит, часть программы работают в многозадачном режиме. Следовательно, они выполняются независимо и одновременно. При этом может возникнуть ситуация, когда два потока попытаются одновременно использовать ресурс, которым в конкретный момент может пользоваться только один поток. Для решения этой проблемы можно создать *блок критического кода,* который будет выполняться в конкретный момент только одним потоком. Этот блок создается с помощью ключевого слова `lock`. Общая форма его синтаксиса выглядит так:

```
Lock(obj) {
   // Блок критического кода.
}
```

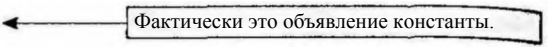
Здесь элемент `obj` — это объект, который стремится получить доступ к коду, определенному в блоке критического кода. Если один поток уже выполняет данный блок кода, второй поток будет ожидать выполнения первого потока. Только после того, как первый поток закончит выполнение этого блока, доступ к нему получит второй поток.

Поле `readonly`

В классе можно создать поле, предназначенное только для чтения, объявив его как `readonly`. Поле, объявленное с указанием ключевого слова `readonly`, может быть

инициализировано, но после этого его невозможно изменить. Следовательно, объявление поля как `readonly` — это хороший способ создания констант (например, для значений размерностей массива, используемых во всей программе). Поля `readonly` могут быть созданы как с модификатором `static`, так и без него.

```
// В программе демонстрируется использование ключевого слова readonly.
using System;

class MyClass {
  public static readonly int SIZE = 10;
}

class DemoReadOnly {
  public static void Main() {
    int[]nums = new int[MyClass.SIZE];

    for(int i=0; i < MyClass.SIZE; i++)
      nums[i] = i;

    foreach(int i in nums)
      Console.Write(i + " ");

    // MyClass.SIZE = 100; // Ошибка!!!
                           // Значение константы не может быть изменено.
  }
}
```



В начале программы при объявлении константы `MyClass.SIZE` ей присваивается значение 10. После этого ее можно использовать, но нельзя изменять. Вы можете убедиться в этом, если удалите символ комментария перед началом последнего оператора и попробуете компилировать программу. Компилятор сообщит об ошибке.

Ключевое слово `stackalloc`

Выделить память из стековой памяти можно с помощью ключевого слова `stackalloc`. Его можно указывать только во время инициализации локальных переменных. Общая форма синтаксиса ключевого слова `stackalloc` представлена ниже.

```
type *p = stackalloc type[size]
```

Здесь элемент `p` — это указатель, который получает адрес области памяти, достаточно большой для хранения указанного количества объектов типа `type` (количество объектов указывается вместо слова `size`). Ключевое слово `stackalloc` используется в небезопасном коде.

Обычно память для объектов выделяется из *кучи* (динамически распределяемой области памяти), то есть из области свободной памяти. Выделение памяти из стековой памяти является исключением. Переменные, помещенные в стековую память, не перемешаются при «сборке мусора», но существуют только во время выполнения блока, в котором они объявлены. Единственным преимуществом использования ключевого слова `stackalloc` является то, что после выделения памяти подобным способом программисту не нужно беспокоиться о перемещении переменных во время «сборки мусора».

Оператор `using`

Ключевое слово `using` может использоваться не только в качестве *директивы,* но и в качестве *оператора.* В этом случае его синтаксис выглядит так:

```
using (obj) {
    // использование объекта.
}
```

Здесь элемент `obj` — это объект, который используется внутри блока `using`. После выполнения операторов блока вызывается метод `Dispose()`. Оператор `using` применяется только для объектов, которые реализуют интерфейс `System.IDisposable`.

Модификаторы `const` и `volatile`

Модификатор `const` используется для объявления переменных, значение которых нельзя изменить. Этим переменным необходимо присваивать начальные значения при их объявлении. Переменная с модификатором `const` является константой. Оператор

```
const int i = 10;
```

создаст переменную `i` с модификатором `const`, имеющую значение 10. С помощью модификатора `volatile` компилятору сообщается, что значение переменной может изменяться способами, не определенными в программе явно. Например, значение переменной, в которой хранится информация о текущем системном времени, может обновляться автоматически операционной системой. В таком случае содержимое переменной изменяется без явного выполнения оператора присваивания. Причиной, по которой следует учитывать возможность внешнего изменения переменной, является то, что C#-компилятору разрешено оптимизировать некоторые выражения автоматически, предполагая, что содержимое переменной не изменяется, если она не встречается в коде слева от оператора присваивания. Но если процессы за пределами выполняемого кода (например, второй поток выполнения программы) могут изменить значение переменной, то такое предположение, на основе которого компилятором выполняется оптимизация, станет ошибочным. С помощью модификатора `volatile` компилятору сообщается, что он должен получать (считывать) значение этой переменной при каждом обращении к ней.

Что делать дальше

Примите наши поздравления! Если вы прочитали и проработали все предыдущие 12 глав, то смело можете называть себя программистом на C#. Безусловно, многое еще предстоит изучить — библиотеки, подсистемы, но вы имеете базу, используя которую можете расширять свои знания и опыт. Ниже мы перечислим некоторые темы, которые рекомендуем вам изучить в будущем.

- Использование C# для создания Windows-приложений, в частности GUI (графические пользовательские интерфейсы), в которых используются различные элементы GUI, такие как кнопки, меню, списки и полосы прокрутки.
- Создание многопотоковых приложений.
- Встроенные компоненты.
- Использование коллекций классов C#, поддерживающих стеки, очереди и списки.
- Использование классов, предназначенных для работы в сети.
- Управление версиями ваших приложений.

Контрольные вопросы

1. Напишите, как объявить делегат с именем `filter`, который возвращает значение типа `double` и получает один аргумент типа `int`.
2. Как создается многоадресный делегат? Какие существуют ограничения?
3. Какая взаимосвязь существует между делегатами и событиями?
4. Может ли событие быть широковещательным? Экземпляру или классу передается событие?
5. Назовите основное преимущество использования пространства имен.
6. Напишите синтаксис создания псевдонима с помощью оператора `using`.
7. Какие два вида операторов преобразования существуют в C#? Напишите общую форму их синтаксиса.
8. Назовите директивы препроцессора, которые используются для условного компилирования.
9. Напишите синтаксис использования атрибута.
10. Что такое небезопасный код?
11. Как во время выполнения программы определить тип объекта?

Продолжайте самостоятельно экспериментировать с элементами языка C#.

Герберт Шилдт

C#: учебный курс

Перевод с английского под редакцией А.Падалки

Редактор	*Е. Насыроиа*
Художник	*Н. Бирж-акпн*
Корректор	*Е. Курбатова*
Технический редактор	*З.Лобач*

ББК 32.973-018я7
УДК 681.3.06(075)

Шилдт Г.
U57 C#: учебный курс. — СПб.: Питер; К.: Издательская группа BHV, 2003. — 512 с.: ил.

ISBN 966-552-121-7
ISBN 5-94723-167-0

Эта книга послужит прекрасным пособием как для специалистов, желающих изучить язык C# — новый язык фирмы Microsoft, так и для начинающих пользователей, не владеющих in одним из языков программирования. Книга написана простым и доступным языком. Ее автор, Герберт Шилдт. получил мировую известность благодаря изданию серии книг по программированию.

ISBN 966-552-121-7
ISBN 5-94723-167-0
ISBN 0-07-213329-5 (англ.)

ООО «Питер Принт». 196105, Санкт-Петербург, ул. Благодатная, д. 67.
Лицензия ИД № 05784 от 07.09.01.
ООО «Издательская группа BHV»
Свидетельство о занесении в Государственный реестр серия ДК № 175 от 13.09 2<Ю0
Налоговая льгота общероссийский классификатор продукции ОК 005-93, том 2; 953005 литература учебная.
Подписано в печать 31.01.03. Формат 70x100$^{1}/_{16}$. Усл. п. д. 41.28. Доп. тираж 4000 экз. Заказ № 2281.
Отпечатано с фотоформ в ФГУП «Печатный двор» им. А. М. Горького
Министерства РФ по делам печати, телерадиовещания и средств массовых коммуникации.
197)10, Санкт-Петербург, Чкаловский пр., 15.