Т.А. Павловская

Программирование на языке высокого уровня C#

ИНТУИТ
НАЦИОНАЛЬНЫЙ ОТКРЫТЫЙ УНИВЕРСИТЕТ

Программирование на языке высокого уровня C#

2-е издание, исправленное

Павловская Т.А.

Национальный Открытый Университет "ИНТУИТ"
2016

Программирование на языке высокого уровня C#/ Т.А. Павловская - М.: Национальный Открытый Университет "ИНТУИТ", 2016

Задача этого курса — кратко, доступно и строго изложить основы C#, одного из самых перспективных современных языков программирования. Курс предназначен для изучающих язык "с нуля", но будет полезен и опытным программистам, желающим освоить новый язык, не тратя времени на увесистые переводные фолианты.

Курс начинается с краткого введения в платформу .NET, далее описываются простейшие средства языка C#: встроенные типы данных, управляющие конструкции, массивы и строки. Основным понятиям объектно-ориентированного программирования и их реализации в языке C# посвящена вторая половина курса. Читатель познакомится с основными элементами класса, с видами классов и их взаимоотношений. Описываются интерфейсы, делегаты, события, дается введение в структуры данных и основные коллекции библиотеки .NET. Изложение сопровождается простыми примерами. Описание языка соответствует версии C# 2.0 (2005).

Начальные сведения

Первое представление об основных понятиях объектно-ориентированного программирования, платформе .NET и входящей в ее состав среде разработки Visual Studio .NET.

Предисловие

Этот курс лекций построен на основе учебника автора "C#. Программирование на языке высокого уровня", который выпускается издательством ПИТЕР с 2007 года. Учебник имеет гриф Министерства образования Российской федерации и удостоен в 2010 году премии Правительства Санкт-Петербурга "За выдающиеся достижения в сфере высшего и профессионального образования" в составе учебно-методического комплекса по языкам программирования.

В этот комплекс входят также учебники и практикумы по языкам C/C++ и Паскаль, построенные по единому принципу. Соответствующие учебные курсы можно найти на этом сайте. В комплекс входит более 250 индивидуальных вариантов заданий на лабораторные работы в расчете на учебную группу из 20 человек (все варианты можно найти в учебнике [4]) и более 1000 тестовых вопросов. Преподавателям будут полезны презентации лекций. На сайте интернет-школы программирования ссылка: http://ips.ifmo.ru - http://ips.ifmo.ru можно проверить правильность выполнения некоторых лабораторных работ с помощью системы автоматического тестирования программ.

Доброжелательную и конструктивную критику, а также предложения по улучшению курса направляйте автору по адресу pta-ipm@yandex.ru.

Знакомство с платформой .NET и средой Visual Studio .NET

Презентацию к данной лекции Вы можете скачать ссылка: здесь - http://old.intuit.ru/department/pl/phlcsharp/1/csharp01.ppt.

Совокупность средств, с помощью которых программы пишут, корректируют, преобразуют в машинные коды, отлаживают и запускают,

называют средой разработки, или оболочкой. Среда разработки обычно содержит:

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

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

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

Платформа .NET является открытой средой. Это значит, что компиляторы для нее могут поставляться и сторонними разработчиками. Для обеспечения переносимости компиляторы, входящие в состав платформы, переводят программу не в машинные коды, а в промежуточный язык (Common Intermediate Language, CIL, или просто IL), который не содержит команд, зависящих от языка, операционной системы и типа компьютера. Программа на этом языке выполняется не самостоятельно, а под управлением системы, которая называется общеязыковой средой выполнения (Common Language Runtime, CLR).

Среда CLR может быть реализована для любой операционной системы. При выполнении программы CLR вызывает так называемый JIT-компилятор, переводящий код с языка IL в машинные команды

конкретного процессора, которые немедленно выполняются. JIT означает "just in time", что можно перевести как "вовремя", то есть компилируются только те части программы, которые требуется выполнить в данный момент. Каждая часть программы компилируется один раз и сохраняется в кэше[1)] для дальнейшего использования.

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

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

Платформа .NET содержит огромную библиотеку классов, которые можно использовать при программировании на любом языке .NET. Подробное изучение библиотеки классов .NET — необходимая, но и наиболее трудоемкая задача программиста при освоении этой платформы. Библиотека классов вместе с CLR образуют каркас (framework), то есть основу платформы.

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

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

- Windows-приложение использует элементы интерфейса Windows, включая формы, кнопки, флажки и прочее;
- консольное приложение выполняет вывод "на консоль", то есть в окно командного процессора;
- библиотека классов объединяет классы, которые предназначены для использования в других приложениях;
- веб-приложение — это приложение, доступ к которому выполняется через браузер (например, Internet Explorer) и которое по запросу формирует веб-страницу и отправляет ее клиенту по сети;

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

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

Введение в объектно-ориентированное программирование

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

Понятие объекта в программе совпадает с обыденным смыслом этого слова: объект представляется как совокупность данных, характеризующих его состояние, и функций их обработки, моделирующих его поведение. Вызов функции на выполнение часто называют посылкой сообщения объекту. Например, вызов функции "повернуть руль" интерпретируется как посылка сообщения "автомобиль, поверни руль!".

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

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

Важным свойством объекта является его обособленность. Детали реализации объекта, то есть внутренние структуры данных и алгоритмы их обработки, скрыты от пользователя объекта и недоступны для непреднамеренных изменений. Объект используется через его интерфейс — совокупность правил доступа.

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

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

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

Инкапсуляция позволяет изменить реализацию объекта без модификации основной части программы, если его интерфейс остался прежним. Простота модификации является очень важным критерием качества программы: ведь любой программный продукт в течение

своего жизненного цикла претерпевает множество изменений и дополнений.

Кроме того, инкапсуляция позволяет использовать объект в другом окружении и быть уверенным, что он не испортит не принадлежащие ему области памяти, а также создавать библиотеки объектов для применения во многих программах. Инкапсуляция наряду с наследованием и полиморфизмом считаются тремя "китами", на которых стоит ООП.

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

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

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

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

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

Объект, на основании которого строится новый объект, называется родительским объектом, объектом-предком, базовым классом, или

суперклассом, а унаследованный от него объект — потомком, подклассом, или производным классом.

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

Подводя итог сказанному, сформулирую достоинства ООП:

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

Эти преимущества особенно явно проявляются при разработке программ большого объема и классов программ. Однако ничто не дается даром: создание объектно-ориентированной программы представляет собой весьма непростую задачу. Чтобы эффективно использовать готовые объекты из библиотек, необходимо освоить большой объем достаточно сложной информации. Неграмотное же применение ООП способно привести к созданию излишне сложных программ, которые невозможно отлаживать и усовершенствовать.

Понятие класса

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

объектов этого класса, называемых экземплярами класса.

"Классический" класс содержит данные, задающие свойства объектов класса, и функции, определяющие их поведение. В последнее время в класс часто добавляется третья составляющая — события, на которые может реагировать объект класса[2)].

Все классы библиотеки .NET, а также все классы, которые создает программист в среде .NET, имеют одного общего предка — класс `object`, и организованы в единую иерархическую структуру. Внутри нее классы логически сгруппированы в так называемые пространства имен, которые служат для упорядочивания имен классов и предотвращения конфликтов имен: в разных пространствах имена могут совпадать. Пространства имен могут быть вложенными, их идея аналогична знакомой вам иерархической структуре каталогов на компьютере.

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

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

Последнее, о чем необходимо поговорить, прежде чем начать последовательное изучение языка C#, — как создать простейшее приложение в среде Visual Studio.NET. Для того чтобы изучать именно язык программирования, а не сопутствующую информацию, мы будем работать с консольными приложениями. При запуске консольного приложения операционная система создает так называемое консольное окно, через которое идет весь ввод-вывод программы. Внешне это напоминает работу в операционной системе в режиме командной строки, когда ввод-вывод представляет собой поток символов.

Простейшие приемы работы в среде

Для создания проекта следует после запуска Visual Studio.NET в главном меню выбрать команду File → New → Project…. В левой части открывшегося диалогового окна нужно выбрать пункт Visual C# Projects, а правой — пункт Console Application. В поле Name можно ввести имя проекта, а в поле Location — место его сохранения на диске, если заданные по умолчанию значения вас не устраивают. После щелчка на кнопке OK среда создаст решение и проект с указанным именем. Примерный вид экрана приведен на рис. 1.1.

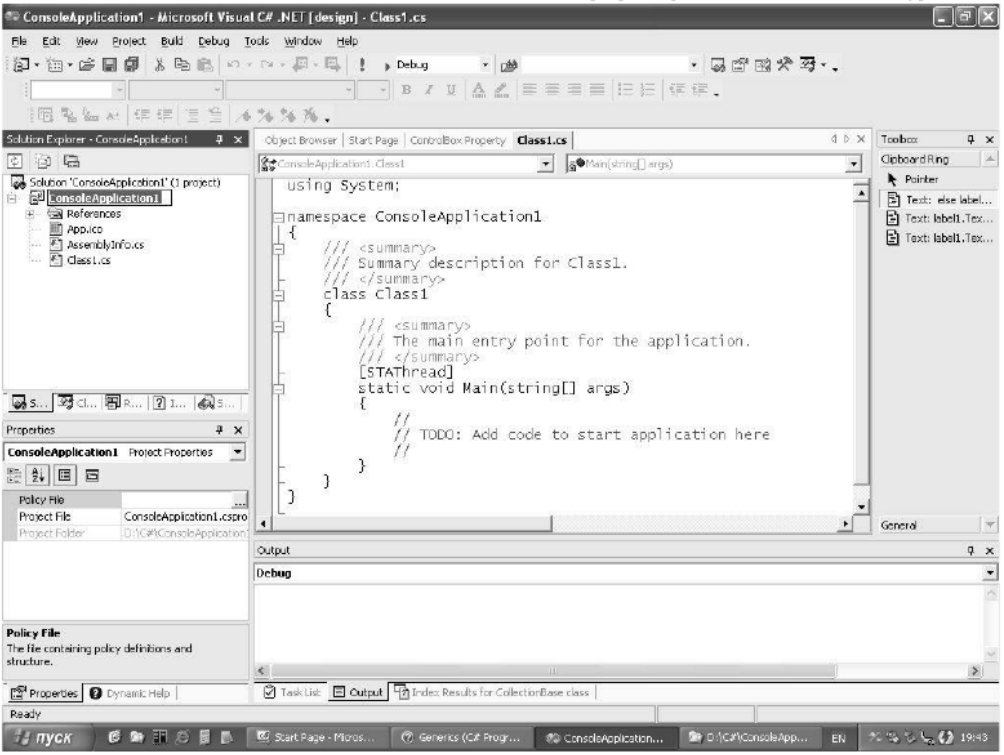


Рис. 1.1. Примерный вид экрана после создания проекта консольного приложения

В верхней части экрана располагается главное меню (с разделами File, Edit, View и т. д.) и панели инструментов (toolbars). В верхней левой части экрана располагается окно управления проектом Solution Explorer (если оно не отображается, следует воспользоваться командой View $\longrightarrow$ Solution Explorer главного меню). В окне перечислены все ресурсы, входящие в проект.

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

В нижней левой части экрана расположено окно свойств Properties (если окна не видно, воспользуйтесь командой View $\longrightarrow$ Properties главного меню). В окне свойств отображаются важнейшие характеристики выделенного элемента. Например, чтобы изменить имя файла, в

котором хранится класс `Class1`, надо выделить этот файл в окне управления проектом и задать в окне свойств новое значение свойства FileName (ввод заканчивается нажатием клавиши Enter).

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

Слева от текста находятся символы структуры: щелкнув на любом квадратике с минусом, можно скрыть соответствующий блок кода. При этом минус превращается в плюс, щелкнув на котором можно опять вывести блок на экран. Это средство хорошо визуально структурирует код и позволяет сфокусировать внимание на нужных фрагментах.

Заготовка консольной программы

Рассмотрим каждую строку заготовки программы (листинг 1.1). В зависимости от версии оболочки вид заготовки может немного варьироваться.

```
using System;

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

```
            //
        }
    }
}
```

Листинг 1.1. Заготовка консольной программы

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

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

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

Внимание

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

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

Упрощенный синтаксис метода:

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

Таким образом, любой метод должен иметь тип, имя и тело, остальные части описания являются необязательными. Среда заботливо поместила внутрь метода `Main` комментарий:

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

Это означает: "Добавьте сюда код, выполняемый при запуске приложения".

Запуск консольного приложения лучше всего выполнять с помощью клавиш Ctrl + F5 (или командой меню Debug → Start Without Debugging). Компилятор может обнаружить в тексте программы синтаксические ошибки. Он сообщает об этом в окне, расположенном в нижней части экрана.

Теперь, когда вы получили общее представление о платформе .NET и основных понятиях ООП, можно приступить к планомерному изучению языка C#.

Вопросы и задания для самостоятельной работы студента

1. Опишите достоинства и недостатки ООП.
2. Изучите состав меню среды разработки.
3. Изучите принципы организации справочной системы.
4. Найдите в справочной системе сведения о том, как создать консольное приложение.
5. Создайте на основе справочной системы консольное приложение, выводящее "Hello, world!", запустите его, изучите структуру и состав каталогов, созданных средой для хранения файлов приложения.

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

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

Состав языка и типы данных

Вводятся базовые для всего дальнейшего изложения понятия: из каких простейших "кирпичиков" состоят все тексты на языке программирования, что понимают под типом данных и какие встроенные типы данных есть в языке C#.

Состав языка

Презентацию к данной лекции Вы можете скачать ссылка: здесь - http://old.intuit.ru/department/pl/phlcsharp/2/csharp02.ppt.

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

Алфавит и лексемы

Все тексты на языке пишутся с помощью его алфавита. В C# используется кодировка символов Unicode. Кодировкой, или кодовой таблицей (character set), называется соответствие между символами и кодирующими их числами. Кодировка Unicode позволяет представить символы всех существующих алфавитов одновременно. Каждому символу соответствует свой уникальный код.

Алфавит C# включает:

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

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

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

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

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

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

Комментарии предназначены для записи пояснений к программе и формирования документации. Правила записи комментариев мы рассмотрим чуть позже.

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

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

```
int a;
```

Это — оператор описания целочисленной переменной `a`.

Идентификаторы

Имена, или идентификаторы, служат для того чтобы обращаться к программным объектам и различать их, то есть идентифицировать. В идентификаторе могут использоваться буквы, цифры и символ подчеркивания. Прописные и строчные буквы различаются, например, `hacker`, `Hacker` и `hAcKeR` — три разных имени.

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

В идентификаторах C# разрешается использовать, помимо латинских букв, буквы национальных алфавитов. Например, правильными являются идентификаторы `Фёкла` и `calc`. Более того, можно применять даже так называемые escape-последовательности Unicode, то есть представлять символ с помощью его кода в шестнадцатеричном виде с префиксом `\u`, например, `\u00F2`.

Примечание

Последняя возможность приведена здесь для полноты картины; не знаю, как вам, а мне трудно себе представить, зачем может понадобиться вставлять в имя шестнадцатеричные коды символов. По современным правилам хорошего стиля программирования имя обязано быть ясным, легко воспринимаемым и при этом как можно более точно отражать смысл и назначение именуемой величины.

Имена даются элементам программы, к которым требуется обращаться: переменным, типам, константам, методам, меткам и т. д. Идентификатор создается на этапе объявления переменной (метода, типа и т. п.), после этого его можно использовать в последующих операторах программы. При выборе идентификатора необходимо следить, чтобы он не совпадал с ключевыми словами.

Ключевые слова

Ключевые слова — это зарезервированные идентификаторы, которые имеют специальное значение для компилятора. Их можно использовать только в том смысле, в котором они определены. Список ключевых слов C# приведен в таблице 2.1. Как видите, их не так уж и много!

Таблица 2.1. Ключевые слова C#

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

Знаки операций и разделители

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

Операции делятся на унарные, бинарные и тернарную по количеству участвующих в них операндов (один, два и три операнда соответственно). Один и тот же знак может интерпретироваться по-

разному в зависимости от контекста.

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

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

Литералы (константы)

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

Описание и примеры констант каждого типа приведены в таблице 2.2. Примеры, иллюстрирующие наиболее часто употребляемые формы констант, выделены полужирным шрифтом.

Таблица 2.2. Константы в C#

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

	следуют шестнадцатеричные цифры (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F), а за цифрами, в свою очередь, может следовать суффикс (U, u, L, l, UL, Ul, uL, ul, LU, Lu, lU, lu)	0xA 0x1B8 0X00FF 0xAU 0x1B8LU 0X00FFl
Вещественная	С фиксированной точкой: [цифры][.][цифры][суффикс] Суффикс — один из символов F, f, D, d, M, m	5.7 .001 35. 5.7F .001d 35. 5F .001f 35m
	С порядком: [цифры][.] [цифры]{E|e}[+|–][цифры] [суффикс] Суффикс — один из символов F, f, D, d, M, m	0.2E6 .11e+3 5E-10 0.2E6D .11e–3 5E10
Символьная	Символ, заключенный в апострофы	'A' 'ю' '*' '\0' '\n' '\xF' '\x74' '\uA81B'
Строковая	Последовательность символов, заключенная в кавычки	"Здесь был Vasia" "\tЗначение r = \xF5 \n" "Здесь был \u0056\u0061" "C:\\temp\\file1.txt" @"C:\temp\file1.txt"
Константа `null`	Ссылка, которая не указывает ни на какой объект	`null`

Логических литералов всего два: `true` и `false`. Они широко используются в качестве признаков наличия или отсутствия чего-либо.

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

шестнадцатеричной системе счисления, а вещественные — только в десятичной системе, но в двух формах: с фиксированной точкой и с порядком. Вещественная константа с порядком представляется в виде мантиссы и порядка. Мантисса записывается слева от знака экспоненты (`E` или `e`), порядок — справа от знака. Значение константы определяется как произведение мантиссы и возведенного в указанную в порядке степень числа 10 (например, 1.3e2 = 1,3 x 10^2 = 130). Пробелы внутри константы не допускаются.

Если требуется сформировать отрицательную целую или вещественную константу, то перед ней ставится знак унарной операции изменения знака (`–`), например, –218.

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

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

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

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

Допустимые значения последовательностей приведены в таблице 2.3.

Таблица 2.3. Управляющие последовательности в C#

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

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

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

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

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

Этот литерал при выводе будет выглядеть так:

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

своим умом

В C# есть и второй вид литералов — дословные (verbatim strings). Эти литералы предваряются символом `@`, который отключает обработку управляющих последовательностей и позволяет получать строки в том виде, в котором они записаны. Чаще всего дословные литералы применяются при задании полного пути файла. Посмотрите, насколько лучше воспринимается второй вариант записи одного и того же пути:

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

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

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

Комментарии

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

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

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

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

Тип данных однозначно определяет:

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

Например, целые и вещественные числа, даже если они занимают одинаковый объем памяти, имеют совершенно разные диапазоны возможных значений.

Каждое выражение в программе имеет определенный тип. Компилятор использует информацию о типе при проверке допустимости описанных в программе действий.

Память, в которой хранятся данные во время выполнения программы, делится на две области: стек (stack) и динамическая область, или хип (heap), чаще называемый кучей (поскольку этот термин мне не нравится, я его использовать не буду). Стек используется для хранения величин, память под которые выделяет компилятор, а в динамической области память резервируется и освобождается во время выполнения программы с помощью специальных команд. Основным местом для хранения данных в C# является хип.

Классификация типов

Типы можно классифицировать по разным признакам. Если принять за основу строение элемента, все типы можно разделить на простые (не имеют внутренней структуры) и структурированные (состоят из элементов других типов). По своему "создателю" типы можно разделить на встроенные (стандартные) и определяемые программистом. По способу хранения значений типы делятся на значимые, или типы-значения, и ссылочные. Рассмотрим в первую очередь встроенные

на встроенные (стандартные) и определяемые программистом. По способу хранения значений типы делятся на значимые, или типы-значения, и ссылочные. Рассмотрим в первую очередь встроенные типы C#.

Встроенные типы

Встроенные типы не требуют предварительного определения. Для каждого типа существует ключевое слово, которое используется при описании переменных, констант и т. д. Встроенные типы C# приведены в таблице 2.4. Они однозначно соответствуют стандартным классам библиотеки .NET, определенным в пространстве имен `System`. Как видно из таблицы, существуют несколько вариантов представления целых и вещественных величин. Программист выбирает тип каждой величины, используемой в программе, с учетом необходимого ему диапазона и точности представления данных.

Таблица 2.4. Встроенные типы C#

Название	Ключевое слово	Тип .NET	Диапазон значений	Описание	Размер, битов
Логический тип	`bool`	`Boolean`	`true`, `false`		
Целые типы	`sbyte`	`SByte`	От –128 до 127	Со знаком	8
	`byte`	`Byte`	От 0 до 255	Без знака	8
	`short`	`Int16`	От –32768 до 32767	Со знаком	16
	`ushort`	`UInt16`	От 0 до 65535	Без знака	16
	`int`	`Int32`	От -2×10^9 до 2×10^9	Со знаком	32
	`uint`	`UInt32`	От 0 до 4×10^9	Без знака	32
	`long`	`Int64`	От -9×10^{18} до 9×10^{18}	Со знаком	64
	`ulong`	`UInt64`	От 0 до 18×10^{18}	Без знака	64

Вещественные	`float`	`Single`	От 1.5×10^{-45} до 3.4×10^{38}	7 цифр	32
	`double`	`Double`	От 5.0×10^{-324} до 1.7×10^{308}	15–16 цифр	64
Финансовый тип	`decimal`	`Decimal`	От 1.0×10^{-28} до 7.9×10^{28}	28–29 цифр	128
Строковый тип	`string`	`String`	Длина ограничена объемом доступной памяти	Строка из Unicode-символов	
Тип `object`	`object`	`Object`	Можно хранить все, что угодно	Всеобщий предок	

Примечание

Вещественные типы и финансовый тип являются знаковыми. Для них в столбце "диапазон значений" приведены абсолютные величины допустимых значений.

Логический, или булев, тип содержит всего два значения: `true` (истина) и `false` (ложь). Их внутреннее представление программиста интересовать не должно. Все целые и вещественные типы вместе с символьным и финансовым можно назвать арифметическими типами.

Внутреннее представление величины целого типа — целое число в двоичном коде. В знаковых типах старший бит числа интерпретируется как знаковый (0 — положительное число, 1 — отрицательное).

Вещественные типы, или типы данных с плавающей точкой, хранятся в памяти компьютера иначе, чем целочисленные. Внутреннее представление вещественного числа состоит из двух частей — мантиссы и порядка, причем каждая часть имеет знак. Длина мантиссы определяет точность числа, а длина порядка — его диапазон. Все вещественные типы могут представлять как положительные, так и отрицательные числа. Чаще всего в программах используется тип

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

Тип `decimal` предназначен для денежных вычислений, в которых критичны ошибки округления. Величины типа `decimal` позволяют хранить 28–29 десятичных разрядов. Тип `decimal` не относится к вещественным типам, у них различное внутреннее представление. Величины денежного типа даже нельзя использовать в одном выражении с вещественными без явного преобразования типа. Использование величин финансового типа в одном выражении с целыми допускается.

Любой встроенный тип C# соответствует стандартному классу библиотеки .NET, определенному в пространстве имен `System`. Имя этого класса приведено в третьем столбце таблицы 2.4. Везде, где используется имя встроенного типа, его можно заменить именем класса библиотеки.

Типы литералов

Литералы (константы) тоже имеют тип. Если значение целого литерала находится внутри диапазона допустимых значений типа `int`, литерал рассматривается как `int`, иначе он относится к наименьшему из типов `uint`, `long` или `ulong`, в диапазон значений которого он входит. Вещественные литералы по умолчанию относятся к типу `double`.

Например, константа `10` относится к типу `int`, хотя для ее хранения достаточно и байта, а константа `2147 483 648` будет определена как `uint`. Для явного задания типа литерала служит суффикс, например, `1.1f`, `1UL`, `1000m`. Явное задание применяется в основном для уменьшения количества неявных преобразований типа, выполняемых компилятором.

Чаще всего типы C# разделяют по способу хранения элементов на типы-значения и ссылочные типы (рис. 2.1). Элементы типов-значений, или значимых типов (value types), представляют собой просто последовательность битов в памяти, необходимый объем которой выделяет компилятор. Иными словами, величины значимых типов хранят свои значения непосредственно. Величина ссылочного типа хранит не сами данные, а ссылку на них (адрес, по которому расположены данные). Сами данные хранятся в хипе.

Внимание

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

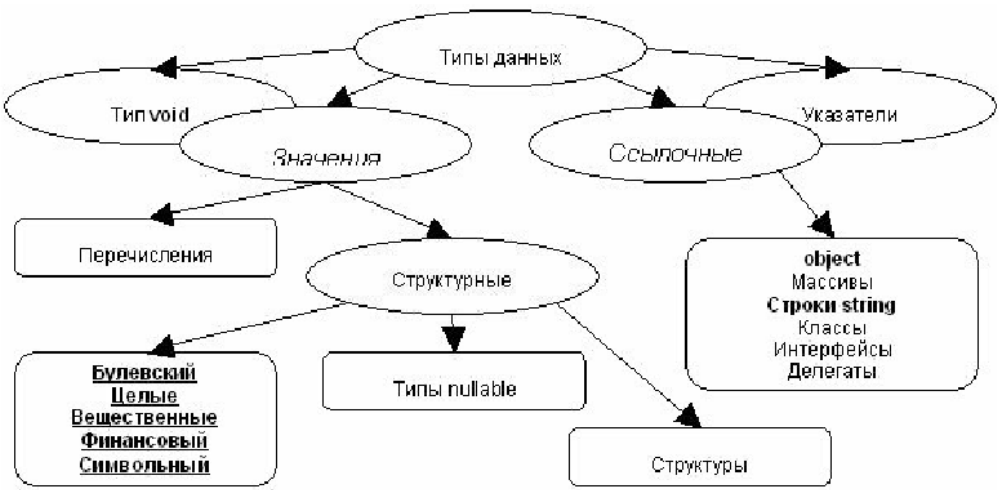


Рис. 2.1. Классификация типов данных C# по способу хранения

рис. 2.2 иллюстрирует разницу между величинами значимого и ссылочного типов. Одни и те же действия над ними выполняются по-разному. Рассмотрим в качестве примера проверку на равенство. Величины значимого типа равны, если равны их значения. Величины ссылочного типа равны, если они ссылаются на одни и те же данные (на рисунке `b` и `c` равны, но `a` не равно `b` даже при одинаковых значениях). Из этого следует, что если изменить значение одной величины ссылочного типа, это может отразиться на другой.

величины ссылочного типа, это может отразиться на другой.

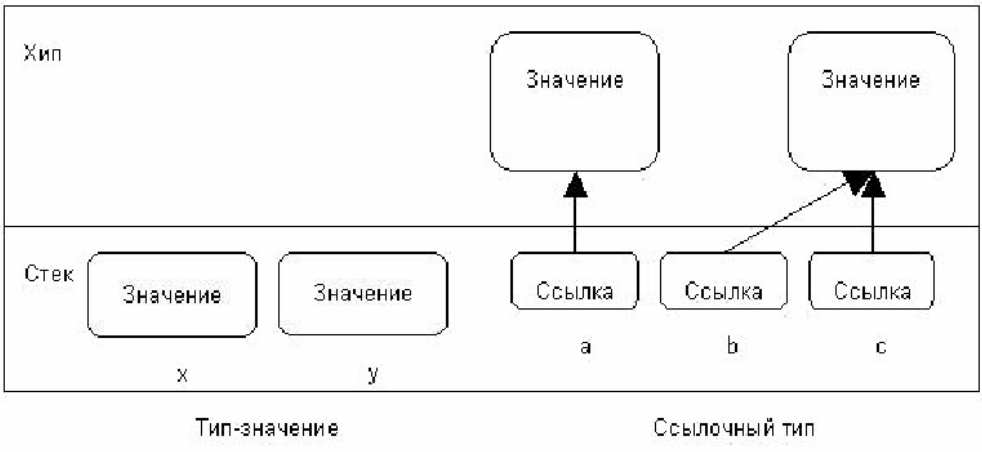


Рис. 2.2. Хранение в памяти величин значимого и ссылочного типов

Упаковка и распаковка

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

Если величина значимого типа используется в том месте, где требуется ссылочный тип, автоматически выполняется создание промежуточной величины ссылочного типа. При необходимости обратного преобразования с величины ссылочного типа "снимается упаковка", и в дальнейших действиях участвует только ее значение.

Вопросы и задания для самостоятельной работы студента

1. Перечислите и опишите лексемы языка C#.
2. Перечислите и опишите литералы языка C#.
3. Перечислите встроенные типы данных и опишите их.

встроенным типам языка, вы знаете?

6. Опишите отличия значимых и ссылочных типов. В чем состоят процессы упаковки и распаковки?

Переменные, операции, выражения

Правила описания переменных и именованных констант, основные операции языка и их приоритеты, правила записи выражений, введение в обработку исключительных ситуаций.

Презентацию к данной лекции Вы можете скачать ссылка: здесь - http://old.intuit.ru/department/pl/phlcsharp/3/csharp03.ppt.

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

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

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

Имя переменной служит для обращения к области памяти, в которой хранится значение переменной. Имя дает программист. Тип переменной выбирается, исходя из диапазона и требуемой точности представления данных. При объявлении можно присвоить переменной некоторое начальное значение, то есть инициализировать ее, например:

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

Здесь описаны:

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

преобразования.

Рекомендуется всегда инициализировать переменные при описании. При инициализации можно использовать не только константу, но и выражение — главное, чтобы на момент описания оно было вычисляемым, например:

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

Программа на C# состоит из классов, внутри которых описывают методы и данные. Структуру программы иллюстрирует рис. 3.1. Переменные, описанные непосредственно внутри класса, называются полями класса. Им автоматически присваивается так называемое "значение по умолчанию" — как правило, это 0 соответствующего типа. Переменные, описанные внутри метода класса, называются локальными переменными. Их инициализация возлагается на программиста.

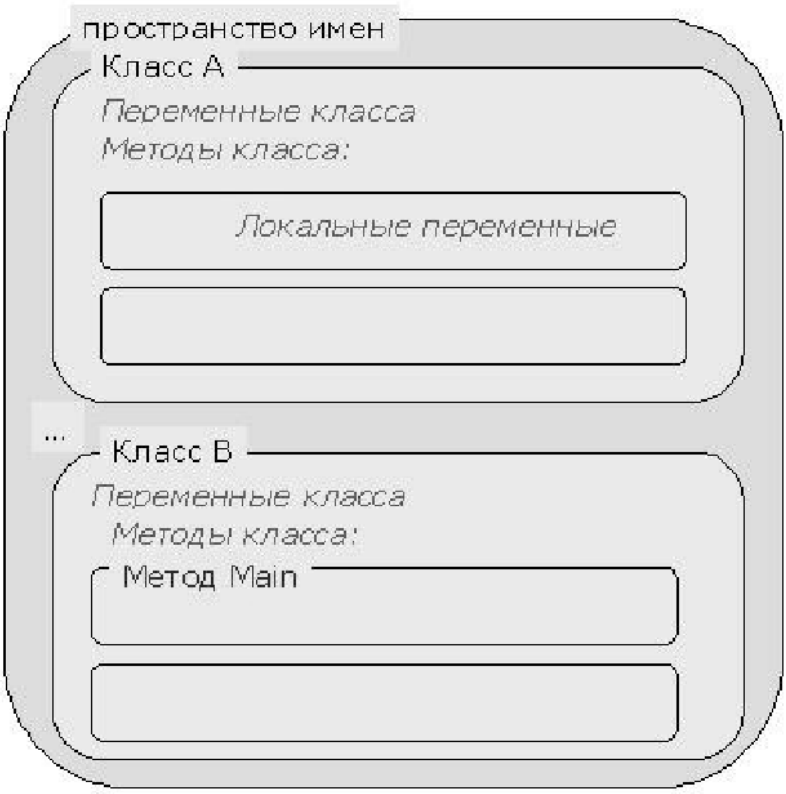


Рис. 3.1. Структура программы

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

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

   void Y()    // ---------------------------------------- метод Y класса X
```

```
    {
      int C;      // локальная переменная C, область действия – метод Y
      int A;      // локальная переменная A (НЕ конфликтует с полем A)

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

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

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

  }         // конец описания класса X
```

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

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

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

```
}
```

Листинг 3.1. Описание переменных

Внимание

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

Именованные константы

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

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

Такие величины называют именованными константами, или просто константами. Они применяются для того, чтобы вместо значений констант можно было использовать в программе их имена. Это делает программу более понятной и облегчает внесение в нее изменений. Именованные константы должны обязательно инициализироваться при описании, например:

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

Операции и выражения

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

Например, `a + 2` — это выражение, в котором `+` является знаком операции, а `a` и `2` — операндами. Пробелы внутри знака операции, состоящей из нескольких символов, не допускаются. Операции C#

приведены в таблице 3.1.

Таблица 3.1. Операции C#

Категория	Знак операции	Название
Первичные	`x()`	Вызов метода или делегата
	`x[]`	Доступ к элементу
	`x++`	Постфиксный инкремент
	`x--`	Постфиксный декремент
	`new`	Выделение памяти
	`typeof`	Получение типа
	`checked`	Проверяемый код
	`unchecked`	Непроверяемый код
Унарные	`+`	Унарный плюс
	`–`	Унарный минус (арифметическое отрицание)
	`!`	Логическое отрицание
	`~`	Поразрядное отрицание
	`++x`	Префиксный инкремент
	`--x`	Префиксный декремент
	`(тип)x`	Преобразование типа
Мультипликативные (типа умножения)	`*`	Умножение
	`/`	Деление
	`%`	Остаток от деления
Аддитивные (типа сложения)	`+`	Сложение
	`–`	Вычитание
Сдвига	`<<`	Сдвиг влево
	`>>`	Сдвиг вправо
Отношения и проверки	`<`	Меньше
	`>`	Больше
	`<=`	Меньше или равно

	is	Проверка принадлежности типу
	as	Приведение типа
Проверки на равенство	==	Равно
	!=	Не равно
Поразрядные логические	&	Поразрядная конъюнкция (И)
	^	Поразрядное исключающее ИЛИ
	|	Поразрядная дизъюнкция (ИЛИ)
Условные логические	&&	Логическое И
	||	Логическое ИЛИ
Условная	?:	Условная операция
Присваивания	=	Присваивание
	*=	Умножение с присваиванием
	/=	Деление с присваиванием
	%=	Остаток от деления с присваиванием
	+=	Сложение с присваиванием
	-=	Вычитание с присваиванием
	<<=	Сдвиг влево с присваиванием
	>>=	Сдвиг вправо с присваиванием
	&=	Поразрядное И с присваиванием
	^=	Поразрядное исключающее ИЛИ с присваиванием
	|=	Поразрядное ИЛИ с присваиванием

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

операции расположены по убыванию приоритетов, уровни приоритетов разделены в таблице горизонтальными линиями.

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

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

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

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

Например, `a + b + c` означает `(a + b) + c`, а `a = b = c` означает `a = (b = c)`.

Преобразования встроенных арифметических типов-значений

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

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

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

Внимание

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

Правила неявного преобразования иллюстрирует рис. 3.2. Если один из операндов имеет тип, изображенный на более низком уровне, чем другой, то он приводится к типу второго операнда при наличии пути между ними. Если пути нет, возникает ошибка компиляции.

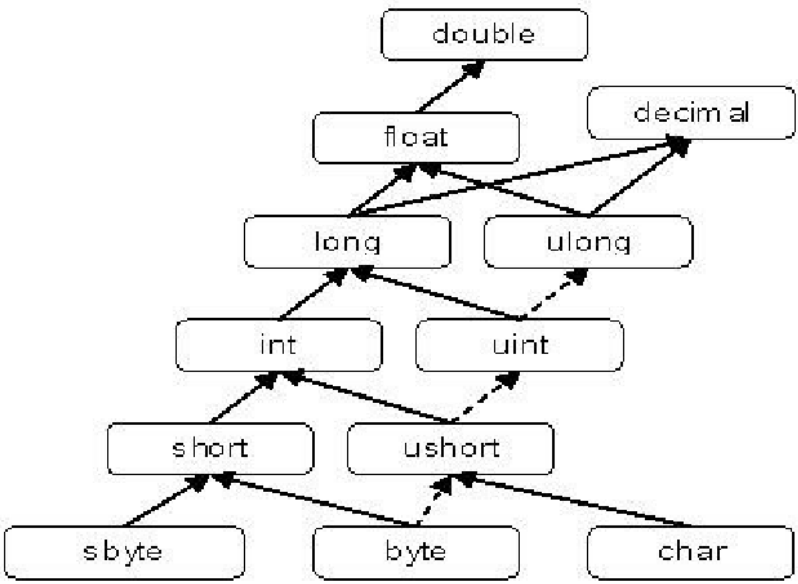


Рис. 3.2. Неявные арифметические преобразования типов

Введение в исключения

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

исключений).

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

Например, при делении на ноль будет сгенерировано исключение `DivideByZeroException`, при недостатке памяти — исключение `OutOfMemoryException`.

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

Справочная информация: Основные операции C#

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

Инкремент и декремент

Операции инкремента (++) и декремента (--) увеличивают и уменьшают операнд на единицу. Они имеют две формы записи — префиксную, когда знак операции записывается перед операндом, и постфиксную. В префиксной форме сначала изменяется операнд, а затем его значение

становится результирующим значением выражения, а в постфиксной форме значением выражения является исходное значение операнда, после чего он изменяется.

Стандартные операции инкремента существуют для целых, символьных, вещественных и финансовых величин.

Операция new

Операция `new` служит для создания нового объекта. Формат операции:

```
new тип ( [ аргументы ] )
```

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

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

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

Операции отрицания

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

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

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

Явное преобразование типа

Операция используется для явного преобразования величины из одного типа в другой. Это требуется в том случае, когда неявного преобразования не существует. При преобразовании из более длинного типа в более короткий возможна потеря информации. Формат операции:

(тип) выражение

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

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

Умножение, деление и остаток от деления

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

Все возможные значения для вещественных операндов приведены в таблице 3.2. Символами `x` и `y` обозначены конечные положительные значения, символом `z` — результат операции вещественного умножения. Если результат слишком велик для представления с помощью заданного типа, он принимается равным значению

"бесконечность", если слишком мал, он принимается за 0. `NaN` (not a number) означает, что результат не является числом.

Таблица 3.2. Результаты вещественного умножения

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

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

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

Если хотя бы один из операндов вещественный, дробная часть результата деления не отбрасывается, а все возможные значения приведены в таблице 3.3.

Таблица 3.3. Результаты вещественного деления

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

+ ∞	+ ∞	- ∞	+ ∞	- ∞	NaN	NaN	NaN
- ∞	- ∞	+ ∞	- ∞	+ ∞	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

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

Операция остатка от деления (`%`) также интерпретируется по-разному для целых, вещественных и финансовых величин. Если оба операнда целочисленные, результат операции вычисляется по формуле `x - (x / y) * y`. Если делитель равен нулю, генерируется исключение `System.DivideByZeroException`.

Если хотя бы один из операндов вещественный, результат операции вычисляется по формуле `x - n * y`, где `n` — наибольшее целое, меньшее или равное результату деления `x` на `y`. Все возможные комбинации значений операндов приведены в таблице 3.4.

Таблица 3.4. Результаты вещественного остатка от деления

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

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

Сложение и вычитание

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

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

Все возможные значения для вещественных операндов приведены в таблице 3.5.

Таблица 3.5. Результаты вещественного сложения

+	y	+0	-0	+ ∞	- ∞	NaN
x	z	x	x	+ ∞	- ∞	NaN
+0	y	+0	+0	+ ∞	- ∞	NaN
-0	y	+0	-0	+ ∞	- ∞	NaN
+ ∞	+ ∞	+ ∞	+ ∞	+ ∞	NaN	NaN
- ∞	- ∞	- ∞	- ∞	NaN	- ∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

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

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

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

Таблица 3.6. Результаты вещественного вычитания

-	y	+0	-0	+ ∞	- ∞	NaN
x	z	x	x	- ∞	+ ∞	NaN
+0	-y	+0	+0	- ∞	+ ∞	NaN
-0	-y	-0	+0	- ∞	+ ∞	NaN
+ ∞	+ ∞	+ ∞	+ ∞	NaN	+ ∞	NaN
- ∞	- ∞	- ∞	- ∞	- ∞	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

Операции сдвига

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

При сдвиге влево (`<<`) освободившиеся разряды обнуляются. При сдвиге вправо (`>>`) освободившиеся биты заполняются нулями, если первый операнд беззнакового типа, и знаковым разрядом в противном случае. Стандартные операции сдвига определены для типов `int`, `uint`, `long` и `ulong`.

Операции отношения и проверки на равенство

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

Таблица 3.7. Результаты операций отношения

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

Поразрядные логические операции

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

При поразрядной конъюнкции (`&`), бит результата равен 1 только тогда, когда соответствующие биты обоих операндов равны 1.

При поразрядной дизъюнкции (`|`), бит результата равен 1 тогда, когда соответствующий бит хотя бы одного из операндов равен 1.

При поразрядном исключающем ИЛИ (⩚) бит результата равен 1 только тогда, когда соответствующий бит только одного из операндов равен 1.

Условные логические операции

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

Результат операции логическое И имеет значение `true`, только если оба операнда имеют значение `true`. Результат операции логическое ИЛИ имеет значение `true`, если хотя бы один из операндов имеет значение `true`. Если значения первого операнда достаточно, чтобы определить результат операции, второй операнд не вычисляется.

Условная операция

Условная операция (? :) имеет три операнда. Ее формат:

```
операнд_1 ? операнд_2 : операнд_3
```

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

Тип результата операции зависит от типа второго и третьего операндов. Если операнды одного типа, он и становится типом результата операции.

Операции присваивания

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

Формат операции простого присваивания (=):

```
переменная = выражение
```

Механизм выполнения операции присваивания такой: вычисляется выражение и его результат заносится в память по адресу, который определяется именем переменной, находящейся слева от знака операции. То, что ранее хранилось в этой области памяти, теряется. Примеры операторов присваивания:

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

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

```
вещественная_переменная := целое_выражение;
```

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

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

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

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

Вопросы и задания для самостоятельной работы студента

1. Где можно описывать переменные? Что входит в описание

переменной?

2. Что происходит при использовании в выражении операндов различных типов? Приведите примеры.
3. Перечислите операции языка C#, сгруппировав их по приоритетам.
4. Что такое NaN? В каких операциях NaN является результатом?
5. К операндам какого типа применимы операции сдвига?
6. Что такое исключительные ситуации?
7. Опишите принципы обработки исключительных ситуаций.

Простейший ввод-вывод. Управляющие операторы

Основные возможности консольного ввода-вывода (класс Console) и управляющие операторы языка (ветвления, циклы, передача управления).

Консольный ввод-вывод

Презентацию к данной лекции Вы можете скачать ссылка: здесь - http://old.intuit.ru/department/pl/phlcsharp/4/csharp04.ppt.

Любая программа при вводе исходных данных и выводе результатов взаимодействует с внешними устройствами. Совокупность стандартных устройств ввода и вывода, то есть клавиатуры и экрана, называется консолью. В языке C# нет операторов ввода и вывода. Вместо них для обмена с внешними устройствами применяются стандартные объекты. Для работы с консолью в C# применяется класс `Console`, определенный в пространстве имен `System`. Методы этого класса `Write` и `WriteLine` уже использовались в листинге 3.1. Поговорим о них подробнее, для чего внесем некоторые изменения в этот листинг. Результаты этих изменений представлены в листинге 4.1.

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

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

Листинг 4.1. Методы вывода

Результат работы программы:

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

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

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

Листинг 4.1 содержит два наиболее употребляемых варианта вызова методов вывода. Если метод `WriteLine` вызван с одним параметром, он может быть любого встроенного типа. В строке, помеченной комментариями "1" и "2", нам требуется вывести в каждой строке не одну, а две величины, поэтому прежде чем передавать их для вывода, их требуется "склеить" в одну строку с помощью операции `+`.

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

Оператор 3 иллюстрирует форматный вывод. В этом случае используется другой вариант метода `WriteLine`. Первым параметром методу передается строковый литерал, содержащий, помимо обычных символов, предназначенных для вывода на консоль, параметры в фигурных скобках, а также управляющие последовательности. Параметры нумеруются с нуля, перед выводом они заменяются

значениями соответствующих переменных в списке вывода.

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

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

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

Преобразование можно выполнить либо с помощью специального класса `Convert`, определенного в пространстве имен `System`, либо с помощью метода `Parse`, имеющегося в каждом стандартном арифметическом классе. В листинге 4.2 используются оба способа.

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

         Console.WriteLine( "Введите символ" );
         char c = (char)Console.Read();                  // 2
         Console.ReadLine();                             // 3
         Console.WriteLine( "c = " + c );

         string buf;        // строка – буфер для ввода чисел

         Console.WriteLine( "Введите целое число" );
         buf = Console.ReadLine();
         int i = Convert.ToInt32( buf );                 // 4
```

```
            Console.WriteLine( i );

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

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

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

Листинг 4.2. Ввод с консоли

Ввод строки выполняется в операторе 1. Длина строки не ограничена, ввод выполняется до символа перевода строки.

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

Оператор 3 считывает остаток строки и никуда его не передает. Это необходимо потому, что ввод данных выполняется через буфер — специальную область оперативной памяти. Фактически данные сначала заносятся в буфер, а затем считываются оттуда процедурами ввода. Занесение в буфер выполняется по нажатию клавиши Enter вместе с ее кодом. Метод `Read`, в отличие от `ReadLine`, не очищает буфер,

поэтому следующий после него ввод будет выполняться с того места, на котором закончился предыдущий.

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

Внимание

При вводе вещественных чисел дробная часть отделяется от целой с помощью запятой (или точки - зависит от региональных настроек).

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

Ввод-вывод в файлы

Часто бывает удобно заранее подготовить исходные данные в текстовом файле и считывать их в программе. Вывод из программы тоже бывает полезно выполнить не на экран, а в текстовый файл. Работа с файлами подробно рассматривается позже, а сейчас приведем лишь образцы для использования в программах. В листинге 4.3 приведена версия программы из листинга 4.1, выполняющая вывод не на экран, а в текстовый файл с именем output.txt. Файл создается в том же каталоге, что и исполняемый файл программы, по умолчанию — ...\ConsoleApplication1\bin\Debug.

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

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

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

Листинг 4.3. Вывод в текстовый файл

Для того чтобы использовать в программе файлы, необходимо:

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

Ввод данных из файла выполняется аналогично. В листинге 4.4 приведена программа, аналогичная листингу 4.2, но ввод выполняется из файла с именем input.txt, расположенного в каталоге D:\C#. Текстовый файл можно создать с помощью любого текстового редактора, удобно использовать Visual Studio.NET.

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

            char  c = (char)f.Read();
            f.ReadLine();
```

```
            Console.WriteLine( "c = " + c );

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

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

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

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

Листинг 4.4. Ввод из текстового файла

Математические функции — класс Math

Самая простая программа, которую можно себе представить (не считая "Hello, World!"), состоит из ввода исходных данных, вычислений по каким-то формулам и вывода результата. В выражениях, из которых состоят формулы, часто используются математические функции, например, синус или возведение в степень. Они реализованы в классе `Math`, определенном в пространстве имен `System`. Описание методов и полей класса приведено в таблице 4.1.

Таблица 4.1. Основные поля и статические методы класса Math

Имя	Описание	Результат	Пояснения
			| x | записывается как

			`Abs(x)`
`Acos`	Арккосинус	`double`	`Acos(double x)`
`Asin`	Арксинус	`double`	`Asin(double x)`
`Atan`	Арктангенс	`double`	`Atan2(double x, d y)` — угол, тангенс ко есть результат делени x
`BigMul`	Произведение	`long`	`BigMul(int x, int`
`Ceiling`	Округление до большего целого	`double`	`Ceiling(double x`
`Cos`	Косинус	`double`	`Cos(double x)`
`Cosh`	Гиперболический косинус	`double`	`Cosh(double x)`
`DivRem`	Деление и остаток	Перегружен	`DivRem(x, y, rem)`
`E`	База натурального логарифма (число e)	`double`	2,71828182845905
`Exp`	Экспонента	`double`	e x записывается как `Exp(x)`
`Floor`	Округление до меньшего целого	`double`	`Floor(double x)`
`IEEERemainder`	Остаток от деления	`double`	`IEEERemainder(d x, double y)`
`Log`	Натуральный логарифм	`double`	$\log_e$ x записывается ка `Log(x)`
`Log10`	Десятичный логарифм	`double`	$\log_{10}$ x записывается к `Log10(x)`
`Max`	Максимум из двух чисел	Перегружен	`Max(x, y)`
`Min`	Минимум из двух чисел	Перегружен	`Min(x, y)`
	Значение числа		

`PI`	Значение числа π	`double`	3,14159265358979
`Pow`	Возведение в степень	`double`	x^y записывается как `Pow(x, y)`
`Round`	Округление	Перегружен	`Round(3.1)` даст в результате 3 `Round (` даст в результате 4
`Sign`	Знак числа	`int`	Аргументы перегруже
`Sin`	Синус	`double`	`Sin(double x)`
`Sinh`	Гиперболический синус	`double`	`Sinh(double x)`
`Sqrt`	Квадратный корень	`double`	$\sqrt{}$ x записывается ка `Sqrt(x)`
`Tan`	Тангенс	`double`	`Tan(double x)`
`Tanh`	Гиперболический тангенс	`double`	`Tanh(double x)`

В листинге 4.5 приведен пример применения методов класса `Math`.

```
using System;
namespace ConsoleApplication1
{    class Class1
   {    static void Main()
      {
         Console.Write( "Введите x: " );
         double x = double.Parse( Console.ReadLine() );
         Console.Write( "Введите y: " );
         double y = double.Parse( Console.ReadLine() );
         Console.WriteLine( "Максимум из x и y : " + Math.Max(x, y) );
         double z = Math.Pow(Math.Sin(x), 2) +  Math.Pow(Math.Sin(y), 2);
         Console.WriteLine( "Сумма квадратов синусов x и y : " + z );
      }
   }
}
```

Листинг 4.5. Применение методов класса Math

Теперь вы знаете достаточно, чтобы писать линейные программы, все операторы в которых выполняются последовательно, один за другим, например: ввод исходных данных, вычисления по формулам, вывод результатов. Для написания программ более сложной структуры необходимо изучить управляющие операторы, которые, как следует из их названия, управляют потоками вычислений.

Выражения, блоки и пустые операторы

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

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

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

Условный оператор if

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

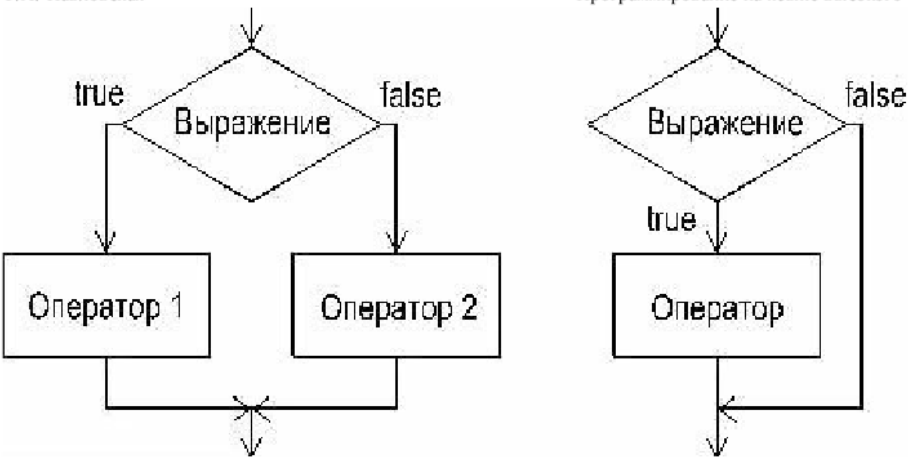


Рис. 4.1. Структурная схема условного оператора

Формат оператора:

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

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

Если в какой-либо ветви требуется выполнить несколько операторов, их необходимо заключить в блок. Блок может содержать любые операторы, в том числе описания и другие условные операторы, но не может состоять из одних описаний.

Примеры условных операторов:

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

Если требуется проверить несколько условий, их объединяют знаками логических условных операций. Например, выражение в примере 2 будет истинно в том случае, если выполнится одновременно условие `a`

< b и одно из условий в скобках. Оператор в примере 3 вычисляет наименьшее значение из трех переменных. Обратите внимание, что компилятор относит часть else к ближайшему ключевому слову if.

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

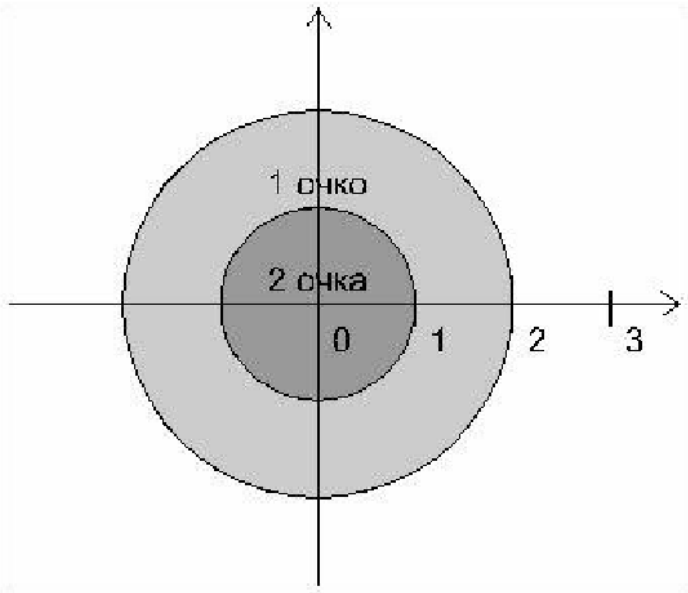


Рис. 4.2. Мишень

Программист выбирает тип переменных, исходя из их назначения. Координаты выстрела нельзя представить целыми величинами, так как это приведет к потере точности результата, а счетчик очков не имеет смысла описывать как вещественный. Программа приведена в листинге 4.6.

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

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

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

Листинг 4.6. Выстрел по мишени

Оператор выбора switch

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

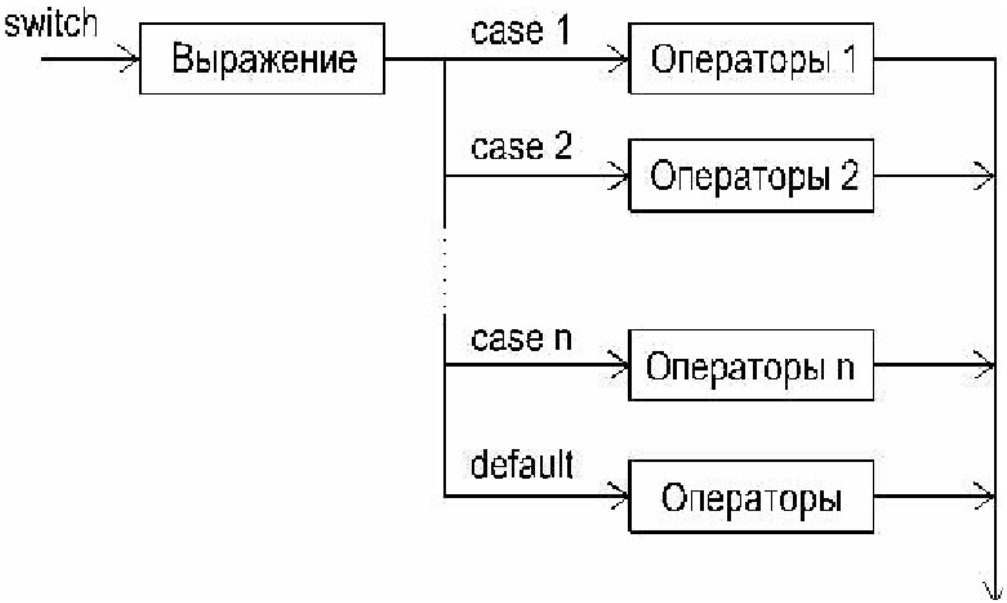


Рис. 4.3. Структурная схема оператора switch

Формат оператора:

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

Выполнение оператора начинается с вычисления выражения. Тип выражения чаще всего целочисленный (включая `char`) или строковый. Затем управление передается первому оператору из списка, помеченному константным выражением, значение которого совпало с вычисленным. Если совпадения не произошло, выполняются операторы, расположенные после слова `default`, а при его отсутствии управление передается следующему за `switch` оператору.

Каждая ветвь переключателя должна заканчиваться явным оператором перехода, а именно одним из операторов `break`, `goto` или `return`:

- оператор `break` выполняет выход из самого внутреннего из объемлющих его операторов `switch`, `for`, `while` и `do` ;
- оператор `goto` выполняет переход на указанную после него метку, обычно это метка `case` одной из нижележащих ветвей оператора `switch` ;
- оператор `return` выполняет выход из функции, в теле которой он записан.

В листинге 4.7 приведен пример программы, реализующей простейший калькулятор на четыре действия.

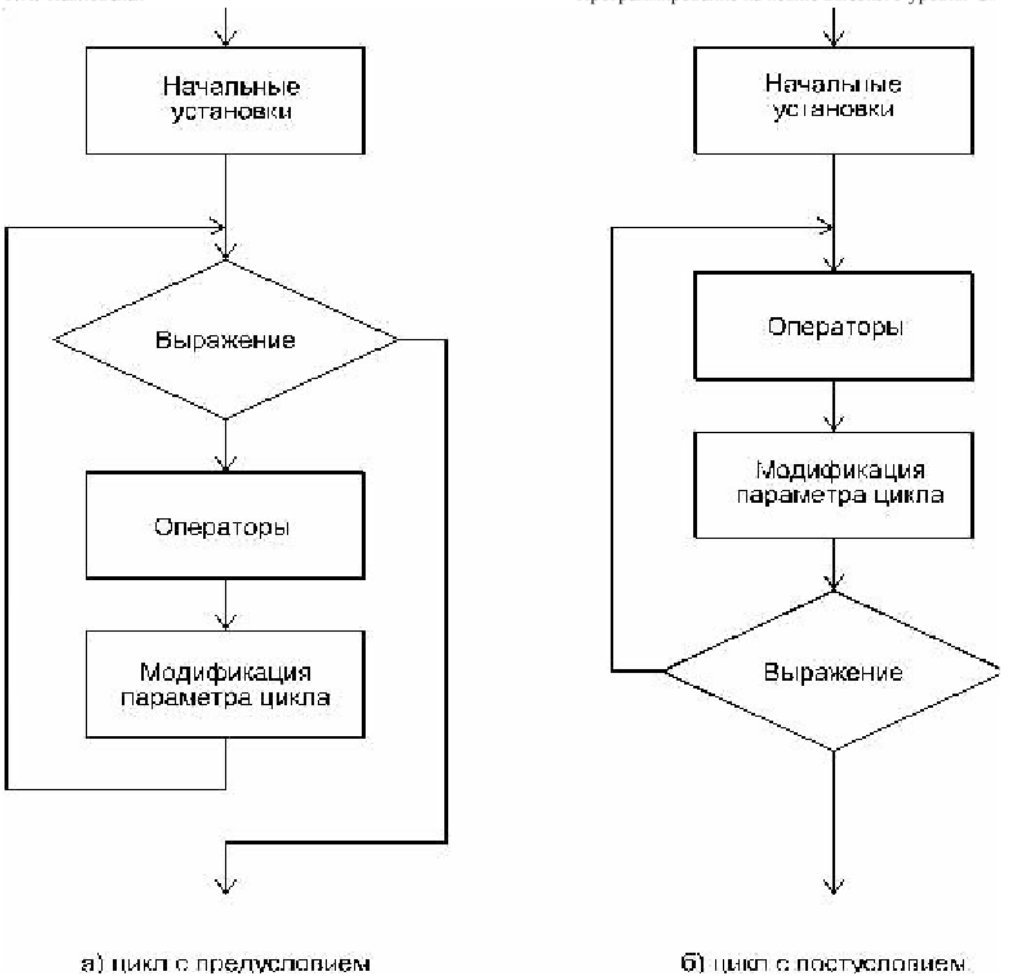
```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            string buf;
            double a, b, res;

            Console.WriteLine( "Введите первый операнд:" );
```

```
            a = double.Parse( Console.ReadLine() );

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

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

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

Листинг 4.7. Калькулятор на четыре действия

Операторы цикла и передачи управления

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



а) цикл с предусловием б) цикл с постусловием

Рис. 4.4. Структурные схемы операторов цикла

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

Проверка условия продолжения цикла выполняется на каждой итерации либо до тела цикла (тогда говорят о цикле с предусловием), либо после тела цикла (цикл с постусловием).

Параметром цикла называется переменная, которая используется при проверке условия продолжения цикла и принудительно изменяется на каждой итерации, причем, как правило, на одну и ту же величину. Если параметр цикла целочисленный, он называется счетчиком цикла.

Цикл завершается, если условие его продолжения не выполняется. Возможно принудительное завершение как текущей итерации, так и цикла в целом. Для этого служат операторы `break`, `continue`, `return` и `goto`. Передавать управление извне внутрь цикла запрещается.

Цикл с предусловием while

Формат оператора:

while (выражение) оператор

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

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

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

$$\mathbf{X} = \begin{pmatrix} x, x < 0 \\ tx, o \leqslant x < 10 \\ 2t, x \geqslant 10 \end{pmatrix}$$

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

Текст программы приведен в листинге 4.8.

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

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

Листинг 4.8. Таблица значений функции с использованием цикла while

Цикл с постусловием do

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

```
do оператор while ( выражение );
```

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

Этот вид цикла применяется в тех случаях, когда тело цикла необходимо

обязательно выполнить хотя бы один раз. Пример программы, выполняющей проверку ввода, приведен в листинге 4.9.

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

Листинг 4.9. Проверка ввода

Цикл с параметром for

Цикл с параметром имеет следующий формат:

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

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

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

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

Выражение типа `bool` определяет условие выполнения цикла: если его

результат равен `true`, цикл выполняется.

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

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

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

Для примера вычислим сумму чисел от 1 до 100:

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

В листинге 4.10 приведена программа, выводящая таблицу значений функции из листинга 4.8.

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

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

Листинг 4.10. Таблица значений функции с использованием

цикла for

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

Цикл `for`:

```
for ( b1; b2; b3 ) оператор;
```

Цикл `while`:

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

Цикл перебора foreach

Цикл `foreach` используется для просмотра всех объектов из некоторой группы данных, например, массива, списка или другого контейнера. Он будет рассмотрен в разделе "Оператор foreach".

Операторы передачи управления

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

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

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

Оператор goto

Оператор безусловного перехода `goto` используется в одной из трех форм:

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

В теле той же функции должна присутствовать ровно одна конструкция вида:

```
метка: оператор;
```

Оператор `goto метка` передает управление на помеченный оператор. Метка — это обычный идентификатор, областью видимости которого является функция, в теле которой он задан. Метка должна находиться в той же области видимости, что и оператор перехода.

Вторая и третья формы оператора `goto` используются в теле оператора выбора `switch`. Оператор `goto case константное_выражение` передает управление на соответствующую константному выражению ветвь, а оператор `goto default` — на ветвь `default`.

Оператор break

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

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

$$y = x - x^3/3! + x^5/5! - x^7/7! + \ldots$$

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

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

Для уменьшения количества выполняемых действий следует воспользоваться рекуррентной формулой получения последующего члена ряда через предыдущий: $C_{n+1} = C_n \times T$, где T — некоторый множитель. Подставив в эту формулу C_n и C_{n+1}, получим выражение для вычисления T:

$$T = 1 + \frac{c_{n+1}}{c_n} + \frac{2n! \cdot x^{2(n+1)}}{x^{2n} \cdot (2(n+1))!} = \frac{x^2}{(2n+1)(2n+2)}$$

В листинге 4.11 приведен текст программы с комментариями.

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            double e = 10^-6;
            const int MaxIter = 500;     // ограничитель количества итераций
            Console.WriteLine( "Введите аргумент:" );
            double x = Convert.ToDouble( Console.ReadLine() );

            bool done = true;                // признак достижения точности
            double ch = x, y = ch;
            for ( int n = 0; Math.Abs(ch) > e; n++ )
            {
                ch *= -x * x / (2 * n + 2 ) / ( 2 * n + 3); // очередной член ряда
                y += ch;                    // добавление члена ряда к сумме
                if ( n > MaxIter ) { done = false; break; }
            }
```

```
            if ( done ) Console.WriteLine( "Сумма ряда - " + y );
            else        Console.WriteLine( "Ряд расходится" );
        }
    }
}
```

Листинг 4.11. Вычисление суммы бесконечного ряда

Получение суммы бесконечного ряда — пример вычислений, которые принципиально невозможно выполнить точно.

Оператор continue

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

Перепишем основной цикл листинга 4.11 с применением оператора `continue`:

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

Оператор return

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

```
return [ выражение ];
```

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

выражение должно отсутствовать.

Базовые конструкции структурного программирования

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

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

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

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

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

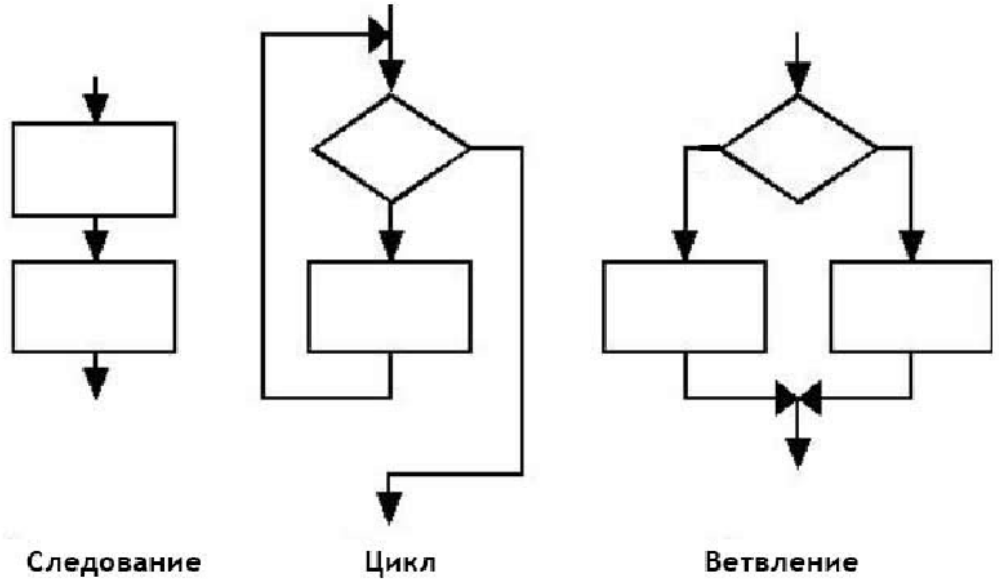


Рис. 4.5. Базовые конструкции структурного программирования

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

Целью использования базовых конструкций является получение программы простой структуры. Такую программу легко читать, отлаживать и при необходимости вносить в нее изменения. В большинстве языков высокого уровня существует несколько реализаций базовых конструкций; в C# им соответствует четыре вида циклов и два вида ветвлений.

Обработка исключительных ситуаций

В языке C# есть операторы, позволяющие обнаруживать и обрабатывать ошибки (исключительные ситуации), возникающие в процессе выполнения программы.

Исключительная ситуация, или исключение, — это возникновение

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

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

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

Исключения генерирует либо среда выполнения, либо программист с помощью оператора `throw`. В таблице 4.2 приведены наиболее часто используемые стандартные исключения, генерируемые средой. Они определены в пространстве имен `System`. Все они являются потомками класса `Exception`, а точнее, потомками его потомка `SystemException`.

Таблица 4.2. Часто используемые стандартные исключения

Имя	Описание
`DivideByZeroException`	Попытка деления на ноль
`FormatException`	Попытка передать в метод аргумент неверного формата
	Индекс массива выходит за

	границы диапазона
`InvalidCastException`	Ошибка преобразования типа
`OutOfMemoryException`	Недостаточно памяти для создания нового объекта
`OverflowException`	Переполнение при выполнении арифметических операций
`StackOverFlowException`	Переполнение стека

Исключения обнаруживаются и обрабатываются в операторе `try`.

Оператор try

Оператор `try` содержит три части:

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

Синтаксис оператора `try`:

try блок [блоки catch] [блок finally]

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

Рассмотрим, каким образом реализуется обработка исключительных ситуаций.

1. Обработка исключения начинается с появления ошибки. Функция

1. Обработка исключения начинается с появления ошибки. Функция или операция, в которой возникла ошибка, генерирует исключение.
2. Выполнение текущего блока прекращается, отыскивается соответствующий обработчик исключения, и ему передается управление.
3. Выполняется блок `finally`, если он присутствует.
4. Если обработчик не найден, вызывается стандартный обработчик исключения. Обычно он выводит на экран окно с информацией об исключении и завершает текущий процесс.

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

Существуют три формы записи обработчиков:

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

Первая форма применяется, когда имя параметра используется в теле обработчика для выполнения каких-либо действий.

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

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

Пример:

```
try {
  ...  // Контролируемый блок
```

```
catch ( OverflowException e ) {
    ...    // Обработка исключений класса OverflowException (переполнени
}
catch ( DivideByZeroException ) {
    ... // Обработка исключений класса DivideByZeroException (деление на
}
catch {
    ...    // Обработка всех остальных исключений
}
```

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

В любом случае, произошло исключение или нет, управление передается в блок завершения `finally` (если он существует), а затем — первому оператору, находящемуся непосредственно за оператором `try`.

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

Оператор throw

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

Синтаксис оператора `throw`:

```
throw [ выражение ];
```

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

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

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

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

Вопросы и задания для самостоятельной работы студента

1. Какой класс используется для консольного ввода-вывода? Приведите примеры ввода и вывода переменных различных типов.
2. Какие средства языка используются для преобразования величин из символьной формы представления во внутреннюю?
3. В каких случаях оператор `switch` предпочтительнее оператора `if`?
4. Какой вид цикла лучше использовать при вводе данных?
5. Поясните смысл использования базовых конструкций структурного программирования.
6. Изучите по справочной системе состав пространства имен System.
7. Изучите свойства и методы стандартного класса Console по справочной системе.

Лабораторные работы

Лабораторная работа 1. Линейные программы

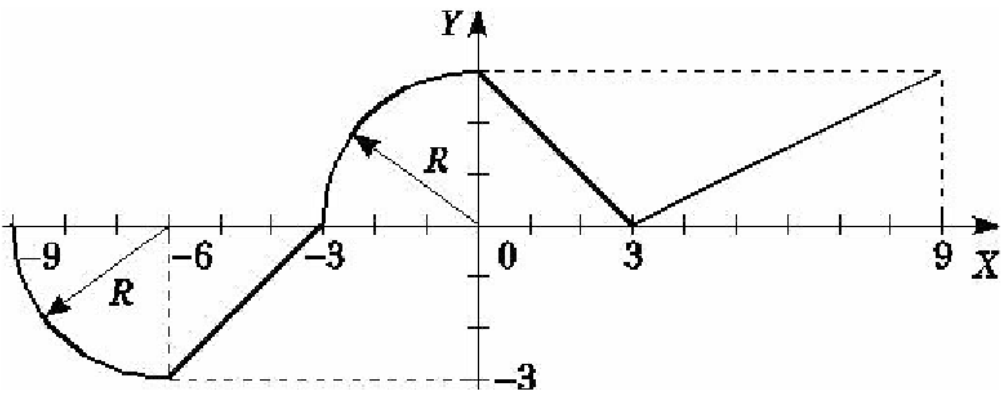
Напишите программу для расчета по двум формулам. Предварительно подготовьте тестовые примеры с помощью калькулятора.

$$z_1 = 2sin^2(3\pi - 2a)cos^2(5\pi + 2a)$$
$$z_2 = \frac{1}{4} - \frac{1}{4}sin(\frac{5}{2}\pi - 8a)$$

Лабораторная работа 2. Разветвляющиеся вычислительные процессы

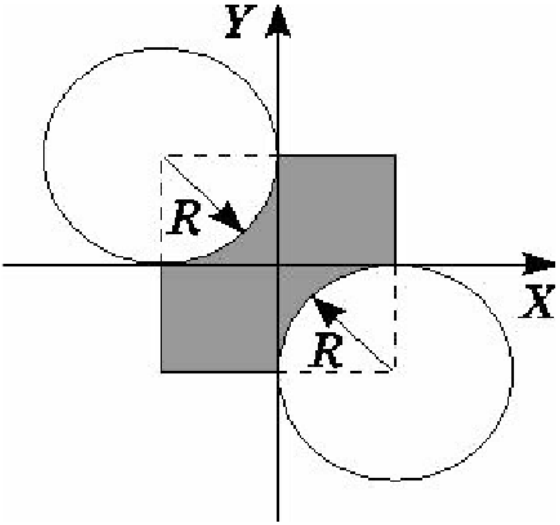
Задание 1. Вычисление значения функции

Написать программу, которая по введенному значению аргумента вычисляет значение функции, заданной в виде графика:



Задание 2. Попадание точки в заштрихованную область

Написать программу, которая определяет, попадает ли точка с заданными координатами в область, закрашенную на рисунке серым цветом. Результат работы программы вывести в виде текстового сообщения.



Лабораторная работа 3. Организация циклов

Теоретический материал: глава 4, разделы "Операторы цикла", "Базовые конструкции структурного программирования".

Задание 1. Таблица значений функции

Вычислить и вывести на экран в виде таблицы значения функции, заданной графически (см. задание 1 лабораторной работы 2), на интервале от $x_{нач}$ до $x_{кон}$ с шагом d x. Интервал и шаг задать таким образом, чтобы проверить все ветви программы. Таблицу снабдить заголовком и шапкой.

Задание 2. Серия выстрелов по мишени

Для десяти выстрелов, координаты которых задаются с клавиатуры, вывести текстовые сообщения о попадании в мишень из задания 2 лабораторной работы 2.

Задание 3. Ряды Тейлора

Вычислить и вывести на экран в виде таблицы значения функции, заданной с помощью ряда Тейлора, на интервале от $x_{нач}$ до $x_{кон}$ с шагом d x с точностью ε. Таблицу снабдить заголовком и шапкой. Каждая строка таблицы должна содержать значение аргумента, значение функции и количество просуммированных членов ряда.

$$ln\frac{x+1}{x-1} = 2\sum_{n=0}^{\infty}\frac{1}{(2n+1)x^{2n+1}} = 2(\frac{1}{x} + \frac{1}{3x^3} + \frac{1}{5x^5} + \cdots)\ |x| > 1$$

Классы: основные понятия

Основные элементы класса: поля, методы, конструкторы, свойства. Виды параметров методов.

Описание класса

Презентацию к данной лекции Вы можете скачать ссылка: здесь - http://old.intuit.ru/department/pl/phlcsharp/5/csharp05.ppt.

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

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

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

Обязательными являются только ключевое слово `class`, имя и тело класса. Тело класса — это список описаний его элементов, заключенный в фигурные скобки. Список может быть пустым, если класс не содержит ни одного элемента. Таким образом, простейшее описание класса может выглядеть так:

```
class Demo {}
```

Спецификаторы определяют свойства класса, а также доступность класса для других элементов программы. Возможные значения спецификаторов перечислены в таблице 5.1. Класс можно описывать непосредственно внутри пространства имен или внутри другого класса. В последнем случае класс называется вложенным.

Таблица 5.1. Спецификаторы класса

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

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

Сейчас мы не будем изучать вложенные классы. Для остальных классов допускаются два спецификатора: `public` и `internal` (по умолчанию).

Класс является обобщенным понятием, определяющим характеристики и поведение некоторого множества конкретных объектов этого класса, называемых экземплярами, или объектами, класса. Объекты создаются явным или неявным образом, то есть либо программистом, либо системой. Программист создает экземпляр класса с помощью операции `new`, например:

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

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

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

Поля и методы являются основными элементами класса. Кроме того, в классе можно задавать целую гамму других элементов: свойства, события, индексаторы, операции, конструкторы, деструкторы, а также типы (рис. 5.1).

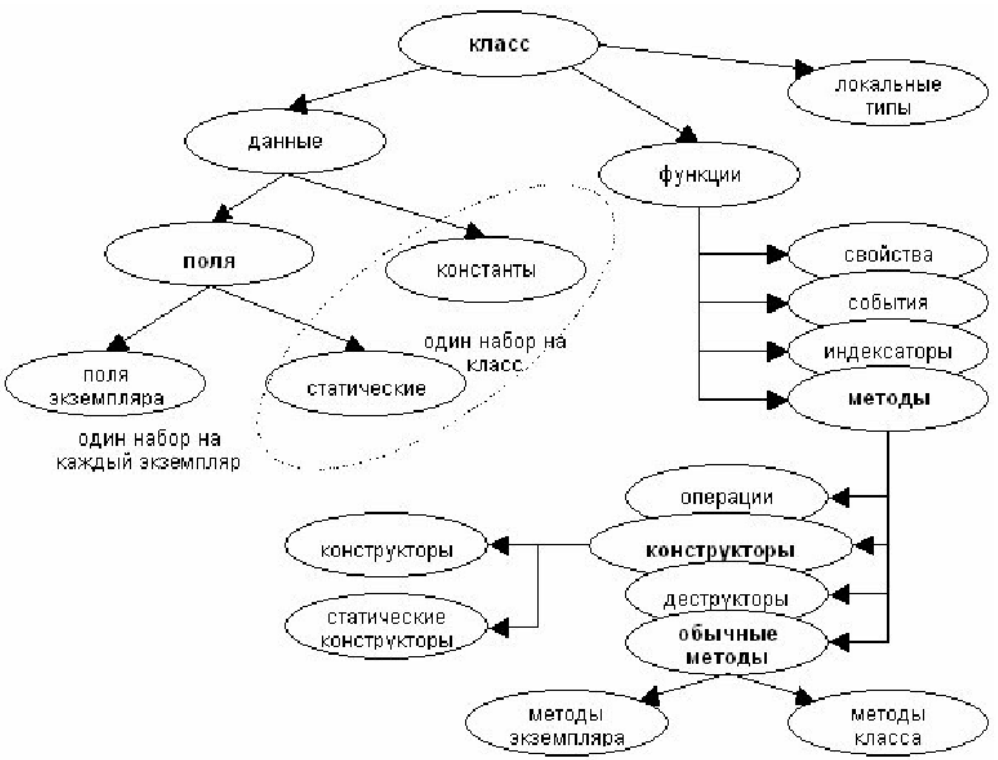


Рис. 5.1. Состав класса

Ниже приведено краткое описание всех элементов класса, изображенных на рисунке:

- Константы класса хранят неизменяемые значения, связанные с классом.
- Методы реализуют вычисления или другие действия, выполняемые классом или экземпляром.
- Свойства определяют характеристики класса в совокупности со способами их задания и получения, то есть методами записи и чтения.
- Конструкторы реализуют действия по инициализации экземпляров или класса в целом.
- Деструкторы определяют действия, которые необходимо выполнить до того, как объект будет уничтожен.
- Индексаторы обеспечивают возможность доступа к элементам класса по их порядковому номеру.
- Операции задают действия с объектами с помощью знаков операций.
- События определяют уведомления, которые может генерировать класс.
- Типы — это типы данных, внутренние по отношению к классу.

Прежде чем начать изучение элементов класса, необходимо поговорить о присваивании и сравнении объектов.

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

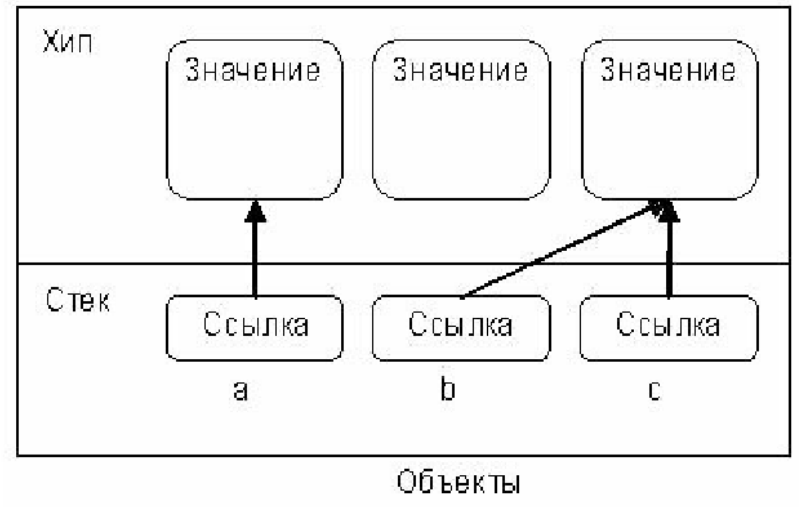


Рис. 5.2. Присваивание объектов

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

Данные: поля и константы

Данные, содержащиеся в классе, могут быть переменными или константами и задаются в соответствии с правилами, рассмотренными в разделах "Переменные" и "Именованные константы". Переменные, описанные в классе, называются полями класса. При описании элементов класса можно также указывать атрибуты и спецификаторы, задающие различные характеристики элементов. Синтаксис описания элемента данных:

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

Возможные спецификаторы полей и констант перечислены в таблице 5.2. Для констант можно использовать только спецификаторы 1–6.

Таблица 5.2. Спецификаторы полей и констант класса

№	Спецификатор	Описание
1	`new`	Новое описание поля, скрывающее унаследованный элемент класса
2	`public`	Доступ к элементу не ограничен
3	`protected`	Доступ только из данного и производных классов
4	`internal`	Доступ только из данной сборки
5	`protected internal`	Доступ только из данного и производных классов и из данной сборки
6	`private`	Доступ только из данного класса
7	`abstract`	Одно поле для всех экземпляров класса
8	`sealed`	Поле доступно только для чтения
9	*`static field`*	Поле может изменяться другим процессом или системой

По умолчанию элементы класса считаются закрытыми (`private`). Для полей класса этот вид доступа является предпочтительным. Все методы класса имеют непосредственный доступ к его закрытым полям.

Внимание

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

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

```
using System;
namespace ConsoleApplication1
{
    class Demo
```

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

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

Листинг 5.1. Класс Demo, содержащий поля и константу

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

Методы

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

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

Синтаксис метода:

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

тело_метода

Первая строка представляет собой заголовок метода. Тело метода, задающее действия, выполняемые методом, чаще всего представляет собой блок.

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

Пример простейшего метода:

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

Тип определяет, значение какого типа вычисляется с помощью метода. Часто употребляется термин "метод возвращает значение". Если метод не возвращает никакого значения, в его заголовке задается тип `void`, а оператор `return` отсутствует.

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

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

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

При этом метод `Sin` возвращает в точку своего вызова вещественное

значение синуса, которое присваивается переменной `y`, а метод `WriteLine` ничего не возвращает. Иллюстрация вызова метода приведена на рис. 5.3.

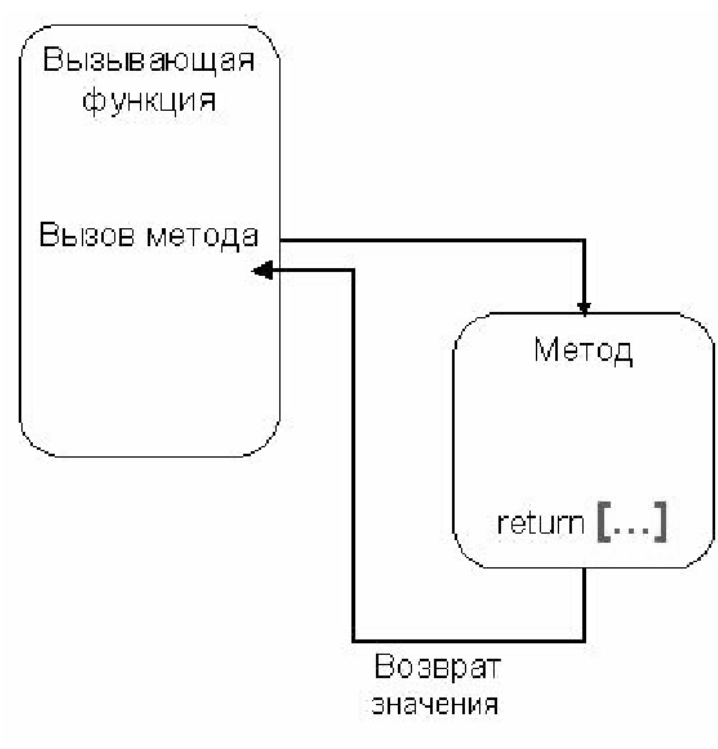


Рис. 5.3. Вызов метода

Внимание

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

Параметры, описываемые в заголовке метода, определяют множество значений аргументов, которые можно передавать в метод. Список аргументов при вызове как бы накладывается на список параметров, поэтому они должны попарно соответствовать друг другу. Для каждого параметра должны задаваться его тип и имя. Например, заголовок метода `Sin` выглядит следующим образом:

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

Имя метода вкупе с количеством, типами и спецификаторами его параметров представляет собой сигнатуру метода. В классе не должно быть методов с одинаковыми сигнатурами.

В листинге 5.2 в класс `Demo` добавлены методы установки и получения значения поля `y`.

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

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

      public void Sety( double y_ )     // метод установки поля y
      {
         y = y_;
      }

      public static string Gets()       // метод получения поля s
      {
         return s;
      }
   }

   class Class1
   {  static void Main()
      {
         Demo x = new Demo();
```

```
            x.Sety(0.12);                // вызов метода установки поля y
            Console.WriteLine(x.Gety());    // вызов метода получения поля y
            Console.WriteLine(Demo.Gets()); // вызов метода получения поля s
        }
    }
}
```

Листинг 5.2. Простейшие методы

Параметры методов

При вызове метода выполняются следующие действия:

1. Вычисляются выражения, стоящие на месте аргументов.
2. Выделяется память под параметры метода в соответствии с их типом.
3. Каждому из параметров сопоставляется соответствующий аргумент (аргументы как бы накладываются на параметры и замещают их).
4. Выполняется тело метода.
5. Если метод возвращает значение, оно передается в точку вызова; если метод имеет тип `void`, управление передается на оператор, следующий после вызова.

При этом проверяется соответствие типов аргументов и параметров и при необходимости выполняется их преобразование. При несоответствии типов выдается диагностическое сообщение. Листинг 5.3 иллюстрирует этот процесс.

```
using System;
namespace ConsoleApplication1
{   class Class1
    {
        static int Max(int a, int b)  // метод выбора максимального значения
        {
            if ( a > b ) return a;
            else         return b;
        }
        static void Main()
```

```
        {
            int a = 2, b = 4;
            int x = Max( a, b );              // вызов метода Max
            Console.WriteLine( x );           // результат: 4

            short t1 = 3, t2 = 4;
            int y = Max( t1, t2 );            // вызов метода Max
            Console.WriteLine( y );           // результат: 4

            int z = Max( a + t1, t1 / 2 * b );  // вызов метода Max
            Console.WriteLine( z );           // результат: 5
        }
    }
}
```

Листинг 5.3. Передача параметров методу

Внимание

Главное требование при передаче параметров состоит в том, что аргументы при вызове метода должны записываться в том же порядке, что и в заголовке метода, и должно существовать неявное преобразование типа каждого аргумента к типу соответствующего параметра. Количество аргументов должно соответствовать количеству параметров. Иллюстрация приведена на рис. 5.4.

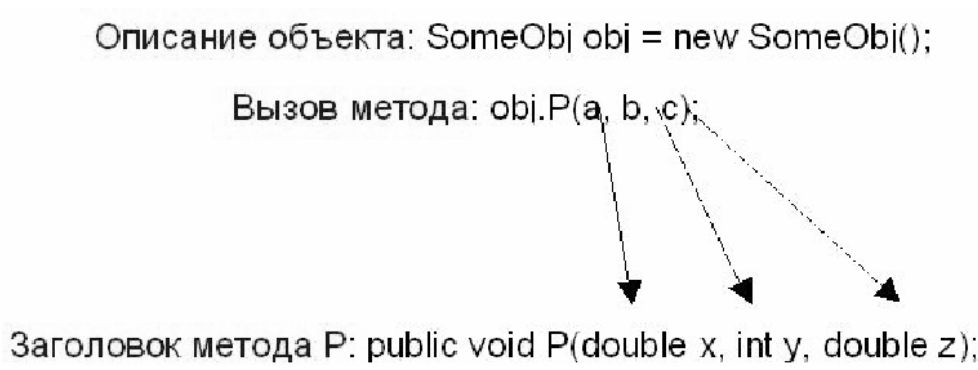


Рис. 5.4. Соответствие параметров при вызове метода

Существуют два способа передачи параметров: по значению и по ссылке.

При передаче по значению метод получает копии значений аргументов, и операторы метода работают с этими копиями. Доступа к исходным значениям аргументов у метода нет, а, следовательно, нет и возможности их изменить.

При передаче по ссылке (по адресу) метод получает копии адресов аргументов, он осуществляет доступ к ячейкам памяти по этим адресам и может изменять исходные значения аргументов, модифицируя параметры.

В C# для обмена данными между вызывающей и вызываемой функциями предусмотрено четыре типа параметров:

- параметры-значения;
- параметры-ссылки — описываются с помощью ключевого слова `ref` ;
- выходные параметры — описываются с помощью ключевого слова `out` ;
- параметры-массивы — описываются с помощью ключевого слова `params`.

Ключевое слово предшествует описанию типа параметра. Если оно опущено, параметр считается параметром-значением. Параметр-массив может быть только один и должен располагаться последним в списке, например:

```
public int Calculate( int a, ref int b, out int c, params int[] d ) …
```

Параметры-значения

Параметр-значение описывается в заголовке метода следующим образом:

```
тип имя
```

Пример заголовка метода, имеющего один параметр-значение целого типа:

```
void P( int x )
```

Имя параметра может быть произвольным. Параметр `x` представляет собой локальную переменную, которая получает свое значение из вызывающей функции при вызове метода. В метод передается копия значения аргумента.

Механизм передачи следующий: из ячейки памяти, в которой хранится переменная, передаваемая в метод, берется ее значение и копируется в специальную область памяти — область параметров. Метод работает с этой копией. По завершении работы метода область параметров освобождается. Этот способ годится только для передачи в метод исходных данных.

При вызове метода на месте параметра, передаваемого по значению, может находиться выражение, для типа которого существует неявное преобразование типа выражения к типу параметра.

Например, пусть в вызывающей функции описаны переменные и им до вызова метода присвоены значения:

```
int    x = 1;
sbyte  c = 1;
ushort y = 1;
```

Тогда следующие вызовы метода `P`, заголовок которого был описан ранее, будут синтаксически правильными:

```
P( x );    P( c );    P( y );    P( 200 );    P( x / 4 + 1 );
```

Параметры-ссылки

Признаком параметра-ссылки является ключевое слово `ref` перед описанием параметра:

```
ref тип имя
```

Пример заголовка метода, имеющего один параметр-ссылку целого типа:

```
void P( ref int x )
```

При вызове метода в область параметров копируется адрес аргумента, и метод через него имеет доступ к ячейке, в которой хранится аргумент. Метод работает непосредственно с переменной из вызывающей функции и, следовательно, может ее изменить, поэтому если в методе требуется изменить значения параметров, они должны передаваться только по ссылке.

Внимание

При вызове метода на месте параметра-ссылки может находиться только ссылка на инициализированную переменную точно того же типа. Перед именем параметра указывается ключевое слово `ref`.

Проиллюстрируем передачу параметров-значений и параметров-ссылок на примере (листинг 5.4).

```
using System;
namespace ConsoleApplication1
{   class Class1
    {
        static void P( int a, ref int b )
        {
            a = 44; b = 33;
            Console.WriteLine( "внутри метода {0} {1}", a, b );
        }
        static void Main()
        {
            int a = 2, b = 4;
            Console.WriteLine( "до вызова     {0} {1}", a, b );
            P( a, ref b );
            Console.WriteLine( "после вызова  {0} {1}", a, b );
        }
    }
}
```

Листинг 5.4. Параметры-значения и параметры-ссылки

Результаты работы этой программы:

```
до вызова     2 4
внутри метода 44 33
после вызова  2 33
```

Несколько иная картина получится, если передавать в метод не величины значимых типов, а экземпляры классов, то есть величины ссылочных типов. Для простоты можно считать, что объекты всегда передаются по ссылке.

Выходные параметры

Довольно часто возникает необходимость в методах, которые формируют несколько величин. В этом случае становится неудобным ограничение параметров-ссылок: необходимость присваивания значения аргументу до вызова метода. Это ограничение снимает спецификатор `out`. Параметру, имеющему этот спецификатор, должно быть обязательно присвоено значение внутри метода.

Изменим описание второго параметра в листинге 5.4 так, чтобы он стал выходным (листинг 5.5).

```
using System;
namespace ConsoleApplication1
{   class Class1
    {
        static void P( int a, out int b )
        {
            a = 44; b = 33;
            Console.WriteLine( "внутри метода {0} {1}", a, b );
        }
        static void Main()
        {
            int a = 2, b;
            P( a, out b );
            Console.WriteLine( "после вызова  {0} {1}", a, b );
        }
    }
}
```

Листинг 5.5. Выходные параметры

При вызове метода перед соответствующим параметром тоже указывается ключевое слово `out`.

Ключевое слово this

Каждый объект содержит свой экземпляр полей класса. Методы находятся в памяти в единственном экземпляре и используются всеми объектами совместно, поэтому необходимо обеспечить работу методов нестатических экземпляров с полями именно того объекта, для которого они были вызваны. Для этого в любой нестатический метод автоматически передается скрытый параметр `this`, в котором хранится ссылка на вызвавший функцию экземпляр. Этот процесс иллюстрирует рис. 5.5.

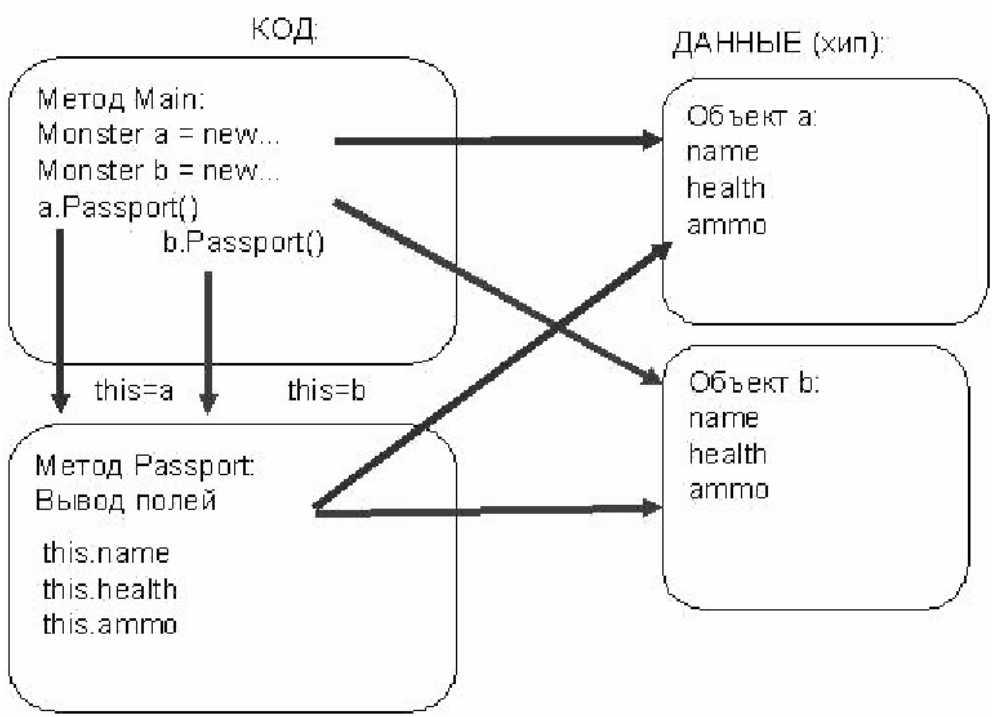


Рис. 5.5. Передача методу скрытого параметра this

В явном виде параметр `this` применяется для того, чтобы возвратить

из метода ссылку на вызвавший объект, а также для идентификации поля в случае, если его имя совпадает с именем параметра метода, например:

```
class Demo
    {
        double y;
        public Demo T()         // метод возвращает ссылку на экземпляр
        {
            return this;
        }
        public void Sety( double y )
        {
            this.y = y;         // полю y присваивается значение параметра y
        }
    }
```

Конструкторы

Конструктор предназначен для инициализации объекта. Он вызывается автоматически при создании объекта класса с помощью операции `new`. Имя конструктора совпадает с именем класса. Ниже перечислены свойства конструкторов

- Конструктор не возвращает значение, даже типа `void`.
- Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации.
- Если программист не указал ни одного конструктора или какие-то поля не были инициализированы, полям значимых типов присваивается нуль, полям ссылочных типов — значение `null`.
- Конструктор, вызываемый без параметров, называется конструктором по умолчанию.

До сих пор мы задавали начальные значения полей класса при описании класса. Это удобно в том случае, когда для всех экземпляров класса начальные значения некоторого поля одинаковы. Если же при создании объектов требуется присваивать полю разные значения, это следует делать в конструкторе. В листинге 5.6 в класс `Demo` добавлен

конструктор, а поля сделаны закрытыми.

```
using System;
namespace ConsoleApplication1
{
   class Demo
   {
      public Demo( int a, double y )          // конструктор с параметрами
      {
         this.a = a;
         this.y = y;
      }

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

      int a;
      double y;
   }

   class Class1
   {   static void Main()
      {
         Demo a = new Demo( 300, 0.002 );     // вызов конструктора
         Console.WriteLine( a.Gety() );       // результат: 0,002
         Demo b = new Demo( 1, 5.71 );        // вызов конструктора
         Console.WriteLine( b.Gety() );       // результат: 5,71
      }
   }
}
```

Листинг 5.6. Класс с конструктором

Часто бывает удобно задать в классе несколько конструкторов, чтобы обеспечить возможность инициализации объектов разными способами. Все конструкторы должны иметь разные сигнатуры.

Если один из конструкторов выполняет какие-либо действия, а другой

должен делать то же самое плюс еще что-нибудь, удобно вызвать первый конструктор из второго. Для этого используется уже известное вам ключевое слово `this` в другом контексте, например:

```
class Demo
    {
        public Demo( int a )                    // конструктор 1
        {
            this.a = a;
        }
        public Demo( int a, double y ) : this( a )   // вызов конструктора 1
        {
            this.y = y;
        }
        ...
    }
```

Конструкция, находящаяся после двоеточия, называется инициализатором.

Как вы помните, все классы в C# имеют общего предка — класс `object`. Конструктор любого класса, если не указан инициализатор, автоматически вызывает конструктор своего предка.

До сих пор речь шла об "обычных" конструкторах, или конструкторах экземпляра. Существует второй тип конструкторов — статические конструкторы, или конструкторы класса. Конструктор экземпляра инициализирует данные экземпляра, конструктор класса — данные класса.

Статический конструктор не имеет параметров, его нельзя вызвать явным образом. Система сама определяет момент, в который требуется его выполнить.

Некоторые классы содержат только статические данные и, следовательно, создавать экземпляры таких объектов не имеет смысла. В версию 2.0 введена возможность описывать статический класс, то есть класс с модификатором `static`. Экземпляры такого класса создавать запрещено, и кроме того, от него запрещено наследовать. Все элементы такого класса должны явным образом объявляться с

модификатором `static` (константы и вложенные типы классифицируются как статические элементы автоматически). В листинге 5.7 приведен пример статического класса.

```
using System;
namespace ConsoleApplication1
{
   static class D
   {
      static int a = 200;
      static double b = 0.002;

      public static void Print ()
      {
         Console.WriteLine( "a = " + a );
         Console.WriteLine( "b = " + b );
      }
   }

   class Class1
   {   static void Main()
      {
         D.Print();
      }
   }
}
```

Листинг 5.7. Статический класс (начиная с версии 2.0)

В качестве "сквозного" примера, на котором будет демонстрироваться работа с различными элементами класса, создадим класс, моделирующий персонаж компьютерной игры. Для этого требуется задать его свойства (например, количество щупальцев или наличие гранатомета) и поведение.

```
using System;
namespace ConsoleApplication1
{
   class Monster
   {
```

```
public Monster()
{
   this.name   = "Noname";
   this.health = 100;
   this.ammo   = 100;
}

public Monster( string name ) : this()
{
   this.name = name;
}

public Monster( int health, int ammo, string name )
{
   this.name   = name;
   this.health = health;
   this.ammo   = ammo;
}

public string GetName()
{
   return name;
}

public int GetHealth()
{
   return health;
}

public int GetAmmo()
{
   return ammo;
}

public void Passport()
{
   Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",
                      name, health, ammo );
}
```

```
        string name;              // закрытые поля
        int health, ammo;
    }

    class Class1
    {
        static void Main()
        {
            Monster X = new Monster();
            X.Passport();
            Monster Vasia = new Monster( "Vasia" );
            Vasia.Passport();
            Monster Masha = new Monster( 200, 200, "Masha" );
            Masha.Passport();
        }
    }
}
```

Листинг 5.8. Класс Monster

Результат работы программы:

```
Monster Noname   health = 100 ammo = 100
Monster Vasia    health = 100 ammo = 100
Monster Masha    health = 200 ammo = 200
```

Свойства

Свойства служат для организации доступа к полям класса. Как правило, свойство связано с закрытым полем класса и определяет методы его получения и установки. Синтаксис свойства:

```
[ атрибуты ] [ спецификаторы ] тип имя_свойства
{
   [ get код_доступа ]
   [ set код_доступа ]
}
```

Значения спецификаторов для свойств и методов аналогичны. Чаще всего свойства объявляются со спецификатором `public`. Код доступа представляет собой блоки операторов, которые выполняются при получении (`get`) или установке (`set`) свойства. Может отсутствовать либо часть `get`, либо `set`, но не обе одновременно.

Если отсутствует часть `set`, свойство доступно только для чтения (read-only), если отсутствует часть `get`, свойство доступно только для записи (write-only). В версии C# 2.0 введена возможность задавать разный уровень доступа для частей `get` и `set`.

Пример описания свойств:

```
public class Button: Control
{
   private string caption;     // закрытое поле, с которым связано свойство
   public string Caption {     // свойство
      get {                    // способ получения свойства
         return caption;
      }
      set {                    // способ установки свойства
         if (caption != value) {
            caption = value;
         }
      }
   }
   ...
}
```

Метод записи обычно содержит действия по проверке допустимости устанавливаемого значения, метод чтения может содержать, например, поддержку счетчика обращений к полю.

В программе свойство выглядит как поле класса, например:

```
Button ok = new Button();
ok.Caption = "OK";              // вызывается метод установки свойства
string s = ok.Caption;          // вызывается метод получения свойства
```

При обращении к свойству автоматически вызываются указанные в нем

методы чтения и установки.

Синтаксически чтение и запись свойства выглядят почти как методы. Метод `get` должен содержать оператор `return`. В методе `set` используется параметр со стандартным именем `value`, который содержит устанавливаемое значение.

Добавим в класс `Monster`, описанный в листинге 5.8, свойства, позволяющие работать с закрытыми полями этого класса. Код класса несколько разрастется, зато упростится его использование.

```
using System;
namespace ConsoleApplication1
{
   class Monster
   {   public Monster()
       {
          this.health = 100;
          this.ammo   = 100;
          this.name   = "Noname";
       }

       public Monster( string name ) : this()
       {
          this.name = name;
       }

       public Monster( int health, int ammo, string name )
       {
          this.health = health;
          this.ammo   = ammo;
          this.name   = name;
       }

       public int Health          // свойство Health связано с полем health
       {
          get
          {
             return health;
```

```
        }
        set
        {
            if (value > 0) health = value;
            else           health = 0;
        }
    }

    public int Ammo                // свойство Ammo связано с полем ammo
    {
        get
        {
            return ammo;
        }
        set
        {
            if (value > 0) ammo = value;
            else           ammo = 0;
        }
    }

    public string Name             // свойство Name связано с полем name
    {
        get
        {
            return name;
        }
    }

    public void Passport()
    {
        Console.WriteLine("Monster {0} \t health = {1} ammo = {2}",
                    name, health, ammo);
    }

    string name;                               // закрытые поля
    int health, ammo;
}
```

```
    class Class1
    {   static void Main()
        {
            Monster Masha = new Monster( 200, 200, "Masha" );
            Masha.Passport();
            --Masha.Health;              // использование свойств
            Masha.Ammo += 100;           // использование свойств
            Masha.Passport();
        }
    }
}
```

Листинг 5.9. Класс Monster со свойствами

Результат работы программы:

```
Monster Masha    health = 200 ammo = 200
Monster Masha    health = 199 ammo = 300
```

Вопросы и задания для самостоятельной работы студента

1. Перечислите и опишите элементы класса в C#.
2. Опишите способы передачи параметров в методы.
3. Запишите алгоритм вычисления чисел Фибоначчи с помощью рекурсии.
4. Для чего в классе может потребоваться несколько конструкторов?
5. Как можно вызвать один конструктор из другого? Зачем это нужно?
6. Что такое this? Что в нем хранится, как он используется?
7. Что такое деструктор? Гарантирует ли среда его выполнение?
8. Какие действия обычно выполняются в части set свойства?
9. Может ли свойство класса быть не связанным с его полями?
10. Можно ли описать разные спецификаторы доступа к частям get и set свойства?

Лабораторные работы

Лабораторная работа № 4. Простейшие классы

Разрабатываемый класс должен содержать следующие элементы: скрытые поля, конструкторы с параметрами и без параметров, методы, свойства. Методы и свойства должны обеспечивать непротиворечивый, полный, минимальный и удобный интерфейс класса. При возникновении ошибок должны выбрасываться исключения.

Задание

Описать класс, реализующий десятичный счетчик, который может увеличивать или уменьшать свое значение на единицу в заданном диапазоне. Предусмотреть инициализацию счетчика значениями по умолчанию и произвольными значениями. Счетчик имеет два метода: увеличения и уменьшения, и свойство, позволяющее получить его текущее состояние. При выходе за границы диапазона выбрасываются исключения.

Написать программу, демонстрирующую все разработанные элементы класса.

Массивы, символы и строки

Одномерные и прямоугольные массивы, базовый класс Array. Оператор foreach. Массивы объектов. Работа с символами и строками. Класс String. Форматирование строк.

Массивы

Презентацию к данной лекции Вы можете скачать ссылка: здесь - http://old.intuit.ru/department/pl/phlcsharp/6/csharp06.ppt.

Массив — это ограниченная совокупность однотипных величин. Элементы массива имеют одно и то же имя, а различаются по порядковому номеру (индексу).

Массив в C# относится к ссылочным типам данных, то есть располагается в динамической области памяти, поэтому создание массива начинается с выделения памяти под его элементы. Элементами массива могут быть величины как значимых, так и ссылочных типов (в том числе массивы). Массив значимых типов хранит значения, массив ссылочных типов — ссылки на элементы. Всем элементам при создании массива присваиваются значения по умолчанию: нули для значимых типов и `null` для ссылочных.

На рис. 6.1 представлен массив, состоящий из пяти элементов любого значимого типа, например, `int` или `double`, а рисунок 6.2 иллюстрирует организацию массива из элементов ссылочного типа.

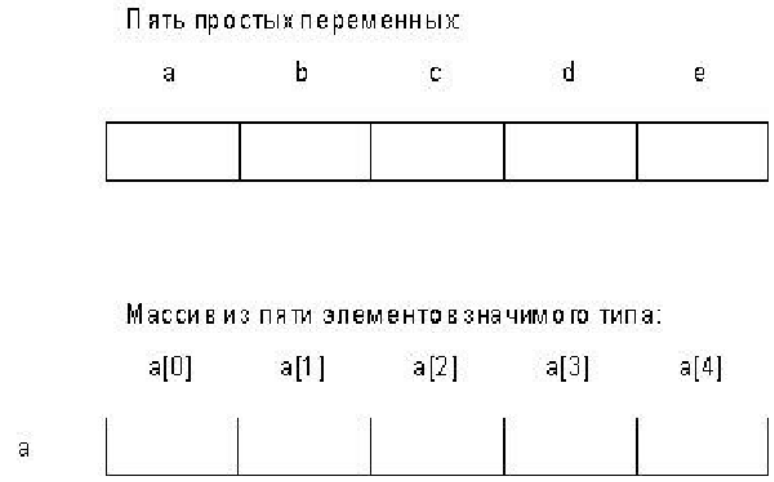


Рис. 6.1. Простые переменные и массив из элементов значимого типа

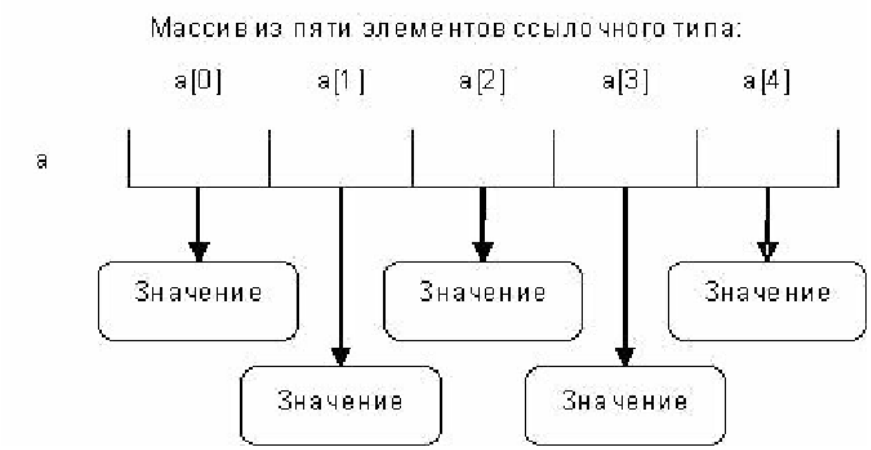


Рис. 6.2. Массив из элементов ссылочного типа

Вот, например, как выглядят операторы создания массива из 10 целых чисел и массива из 100 строк:

```
int[]    w = new int[10];
string[] z = new string[100];
```

В первом операторе описан массив `w` типа `int[]`. Операция `new` выделяет память под 10 целых элементов, и они заполняются нулями.

Во втором операторе описан массив `z` типа `string[]`.Операция `new` выделяет память под 100 ссылок на строки, и эти ссылки заполняются значением `null`. Память под сами строки, составляющие массив, не выделяется — это будет необходимо сделать перед заполнением массива.

Количество элементов в массиве (размерность) не является частью его типа, оно задается при выделении памяти и не может быть изменено впоследствии. Размерность может задаваться не только константой, но и выражением. Результат вычисления этого выражения должен быть неотрицательным, а его тип должен иметь неявное преобразование к `int`, `uint`, `long` или `ulong`.

Элементы массива нумеруются с нуля, поэтому максимальный номер элемента всегда на единицу меньше размерности. Для обращения к элементу массива после имени массива указывается номер элемента в квадратных скобках, например:

```
w[4]      z[i]
```

С элементом массива можно делать все, что допустимо для переменных того же типа. Если при работе с массивом значение индекса выходит за границы массива, генерируется исключение `IndexOutOfRangeException`.

Массивы одного типа можно присваивать друг другу. При этом происходит присваивание ссылок, а не элементов, как и для любого другого объекта ссылочного типа, например:

```
int[] a = new int[10];
int[] b = a;          // b и a указывают на один и тот же массив
```

Все массивы в C# имеют общий базовый класс `Array`, определенный в пространстве имен `System`. В нем есть несколько полезных методов, упрощающих работу с массивами, например, методы получения размерности, сортировки и поиска. Мы рассмотрим эти методы немного позже в разделе "Класс System.Array".

В C# существуют три разновидности массивов: одномерные, прямоугольные и ступенчатые (невыровненные). О последних, менее

распространенных, можно прочитать в учебнике (Павловская Т. А. C#. Программирование на языке высокого уровня. — СПб.: Питер, 2010), а сейчас займемся одномерными массивами.

Одномерные массивы

Одномерные массивы используются в программах чаще всего. Варианты описания массива:

```
тип[] имя;
тип[] имя = new тип [ размерность ];
тип[] имя = { список_инициализаторов };
тип[] имя = new тип [] { список_инициализаторов };
тип[] имя = new тип [ размерность ] { список_инициализаторов };
```

Примеры описаний (один пример на каждый вариант описания):

```
int[] a;                          // 1    элементов нет
int[] b = new int[4];                  // 2    элементы равны 0
int[] c = { 61, 2, 5, -9 };              // 3    new подразумевается
int[] d = new int[] { 61, 2, 5, -9 };      // 4    размерность вычисляется
int[] e = new int[4] { 61, 2, 5, -9 };     // 5    избыточное описание
```

Здесь описано пять массивов. Отличие первого оператора от остальных состоит в том, что в нем фактически описана только ссылка на массив, а память под элементы массива не выделена.

В каждом из остальных массивов по четыре элемента целого типа. Как видно из операторов 3–5, массив при описании можно инициализировать. Если при этом не задана размерность (оператор 3), количество элементов вычисляется по количеству инициализирующих значений. Для полей объектов и локальных переменных можно опускать операцию `new`, она будет выполнена по умолчанию (оператор 2). Если присутствует и размерность, и список инициализаторов, размерность должна быть константой (оператор 4).

В качестве примера рассмотрим программу, которая определяет сумму и количество отрицательных элементов, а также максимальный элемент

массива, состоящего из 6 целочисленных элементов (листинг 6.1).

```
using System;
namespace ConsoleApplication1
{    class Class1
   {
      static void Main()
      {
         const int n = 6;
         int[] a = new int[n] { 3, 12, 5, -9, 8, -4 };

         Console.WriteLine( "Исходный массив:" );
         for ( int i = 0; i < n; ++i )
            Console.Write( "\t" + a[i] );
         Console.WriteLine();

         long sum = 0;              // сумма отрицательных элементов
         int  num = 0;              // количество отрицательных элементов
         for ( int i = 0; i < n; ++i )
            if ( a[i] < 0 )
            {
               sum += a[i];
               ++num;
            }

         Console.WriteLine( "Сумма  отрицательных = " + sum );
         Console.WriteLine( "Кол-во отрицательных = " + num );

         int max = a[0];            // максимальный элемент
         for ( int i = 1; i < n; ++i )
            if ( a[i] > max ) max = a[i];

         Console.WriteLine( "Максимальный элемент = " + max );
      }
   }
}
```

Листинг 6.1. Работа с одномерным массивом

Обратите внимание, что для вывода массива требуется организовать цикл.

Прямоугольные массивы

Прямоугольный массив имеет более одного измерения. Чаще всего в программах используются двумерные массивы. Варианты описания двумерного массива:

```
тип[,] имя;
тип[,] имя = new тип [ разм_1, разм_2 ];
тип[,] имя = { список_инициализаторов };
тип[,] имя = new тип [,] { список_инициализаторов };
тип[,] имя = new тип [ разм_1, разм_2 ] { список_инициализаторов };
```

Примеры описаний (один пример на каждый вариант описания):

```
int[,] a;                              // 1  элементов нет
int[,] b = new int[2, 3];                    // 2  элементы равны 0
int[,] c = {{1, 2, 3}, {4, 5, 6}};           // 3  new подразумевается
int[,] c = new int[,] {{1, 2, 3}, {4, 5, 6}};  // 4  размерность вычисляется
int[,] d = new int[2,3] {{1, 2, 3}, {4, 5, 6}};// 5  избыточное описание
```

К элементу двумерного массива обращаются, указывая номера строки и столбца, на пересечении которых он расположен, например:

```
a[1, 4]      b[i, j]      b[j, i]
```

Внимание

Необходимо помнить, что компилятор воспринимает как номер строки первый индекс, как бы он ни был обозначен в программе.

В качестве примера рассмотрим программу, которая для целочисленной матрицы размером 3 x 4 определяет среднее арифметическое ее элементов и количество положительных элементов в каждой строке.

Для нахождения среднего арифметического элементов массива требуется найти их общую сумму, после чего разделить ее на количество

элементов. Порядок перебора элементов массива (по строкам или по столбцам) роли не играет. Нахождение количества положительных элементов каждой строки требует просмотра матрицы по строкам. Схема алгоритма приведена на рис. 6.3, программа — в листинге 6.2.

```
using System;
namespace ConsoleApplication1
{    class Class1
   {
      static void Main()
      {
         const int m = 3, n = 4;
         int[,] a = new int[m, n] {
            { 2,-2, 8, 9 },
            {-4,-5, 6,-2 },
            { 7, 0, 1, 1 }
          };

         Console.WriteLine( "Исходный массив:" );
         for ( int i = 0; i < m; ++i )
         {
            for ( int j = 0; j < n; ++j )
               Console.Write( "\t" + a[i, j] );
            Console.WriteLine();
         }

         double sum = 0;
         int nPosEl;
         for ( int i = 0; i < m; ++i )
         {
            nPosEl = 0;
            for ( int j = 0; j < n; ++j )
            {
               sum += a[i, j];
               if ( a[i, j] > 0 ) ++nPosEl;
            }
            Console.WriteLine( "В строке {0} {1} положит-х элементов",
               i, nPosEl  );
         }
```

```
            Console.WriteLine( "Среднее арифметическое всех элементов: "
                + sum / m / n );
        }
    }
}
```

Листинг 6.2. Работа с двумерным массивом

Ступенчатые массивы

В ступенчатых массивах количество элементов в разных строках может различаться. В памяти ступенчатый массив хранится иначе, чем прямоугольный: в виде нескольких внутренних массивов, каждый из которых имеет свой размер. Кроме того, выделяется отдельная область памяти для хранения ссылок на каждый из внутренних массивов (рис. 6.3).

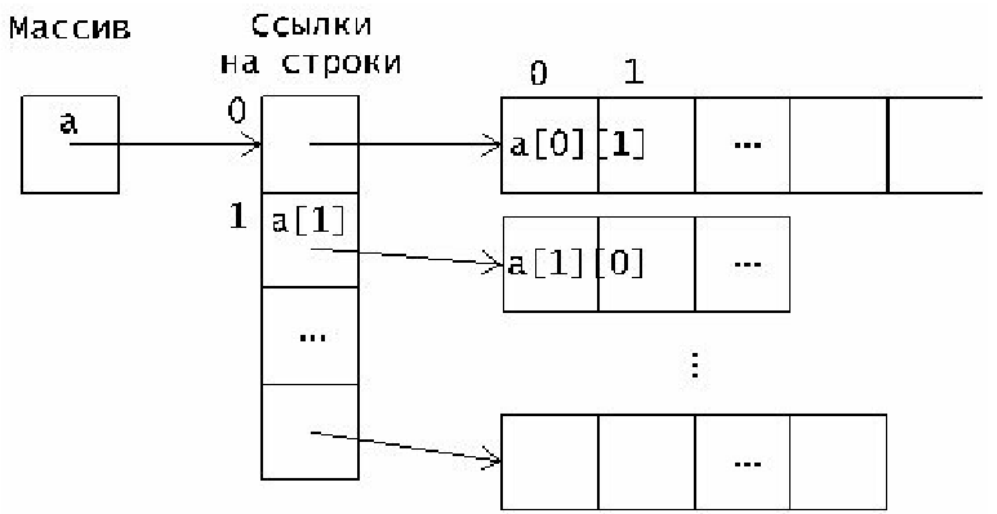


Рис. 6.3. Ступенчатый массив

Фрагмент программы, иллюстрирующий работу со ступенчатым массивом, приведен в листинге 6.5 (после рассмотрения оператора `foreach`).

Класс System.Array

Все массивы в C# построены на основе базового класса `Array`, который содержит полезные для программиста свойства и методы, часть из которых перечислена в таблице 6.1.

Таблица 6.1. Некоторые элементы класса Array

Элемент	Вид	Описание
`Length`	Свойство	Количество элементов массива (по всем размерностям)
`BinarySearch`	Статический метод	Двоичный поиск в отсортированном массиве
`Clear`	Статический метод	Присваивание элементам массива значений по умолчанию
`Copy`	Статический метод	Копирование заданного диапазона элементов одного массива в другой массив
`GetValue`	Метод	Получение значения элемента массива
`IndexOf`	Статический метод	Поиск первого вхождения элемента в одномерный массив
`Reverse`	Статический метод	Изменение порядка следования элементов на обратный
`Sort`	Статический метод	Упорядочивание элементов одномерного массива

В листинге 6.3 продемонстрировано применение элементов класса `Array` при работе с одномерным массивом.

```
using System;
namespace ConsoleApplication1
{    class Class1
   {   static void Main()
       {
          int[] a =  { 24, 50, 18, 3, 16, -7, 9, -1 };
```

```
            PrintArray( "Исходный массив:", a );
            Console.WriteLine( Array.IndexOf( a, 18 ) );

            Array.Sort(a);
            PrintArray( "Упорядоченный массив:", a );
            Console.WriteLine( Array.BinarySearch( a, 18) );
        }

        public static void PrintArray( string header, int[] a )
        {
            Console.WriteLine( header );
            for ( int i = 0; i < a.Length; ++i )
                Console.Write( "\t" + a[i] );
            Console.WriteLine();
        }
    }
}
```

Листинг 6.3. Использование методов класса Array с одномерным массивом

Методы `Sort`, `IndexOf` и *`BinarySearch`* являются статическими, поэтому к ним обращаются через имя класса, а не экземпляра, и передают в них имя массива. Двоичный поиск можно применять только для упорядоченных массивов. В классе `Class1` описан вспомогательный статический метод `PrintArray`, предназначенный для вывода массива на экран.

Результат работы программы:

```
Исходный массив:
        24      50      18      3       16      -7      9       -1
2
Упорядоченный массив:
        -7      -1      3       9       16      18      24      50
5
```

Оператор foreach

Оператор `foreach` применяется для перебора элементов в специальным образом организованной группе данных. Массив является именно такой группой. Удобство этого вида цикла заключается в том, что нам не требуется определять количество элементов в группе и выполнять их перебор по индексу: мы просто указываем на необходимость перебрать все элементы группы. Синтаксис оператора:

```
foreach ( тип имя in выражение ) тело_цикла
```

Имя задает локальную по отношению к циклу переменную, которая будет по очереди принимать все значения из массива выражение (в качестве выражения чаще всего применяется имя массива или другой группы данных). В простом или составном операторе, представляющем собой тело цикла, выполняются действия с переменной цикла. Тип переменной должен соответствовать типу элемента массива.

Например, пусть задан массив:

```
int[] a =  { 24, 50, 18, 3, 16, -7, 9, -1 };
```

Вывод этого массива на экран с помощью оператора `foreach` выглядит следующим образом:

```
foreach ( int x in a ) Console.WriteLine( x );
```

Этот оператор выполняется так: на каждом проходе цикла очередной элемент массива присваивается переменной `x` и с ней производятся действия, записанные в теле цикла.

В листинге 6.4 решается та же задача, что и в листинге 6.1, но с использованием цикла `foreach`. Обратите внимание, насколько понятнее стала программа.

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            int[] a = { 3, 12, 5, -9, 8, -4 };
```

```
            Console.WriteLine( "Исходный массив:" );
            foreach ( int elem in a )
               Console.Write( "\t" + elem );
            Console.WriteLine();

            long sum = 0;              // сумма отрицательных элементов
            int  num = 0;              // количество отрицательных элементов
            foreach ( int elem in a )
               if ( elem < 0 )
               {
                  sum += elem;
                  ++num;
               }

            Console.WriteLine( "sum = " + sum );
            Console.WriteLine( "num = " + num );

            int max = a[0];            // максимальный элемент
            foreach ( int elem in a )
               if ( elem > max ) max = elem;

            Console.WriteLine( "max = " + max );
         }
      }
   }
```

Листинг 6.4. Работа с одномерным массивом с использованием цикла foreach

Внимание

Ограничением оператора `foreach` является то, что с его помощью можно только просматривать значения в группе данных, но не изменять их.

Еще один пример использования оператора для работы со ступенчатым массивом приведен в листинге 6.5.

```
...
int[][] a = new int[3][];
```

```
        a[0] = new int [5] { 24, 50, 18, 3, 16 };
        a[1] = new int [3] { 7, 9, -1 };
        a[2] = new int [4] { 6, 15, 3, 1 };
        Console.WriteLine( "Исходный массив:" );
foreach ( int [] mas1 in a )
          {
            foreach ( int x in mas1 )
                  Console.Write( "\t" + x );
            Console.WriteLine();
          }
       // поиск числа 18 в нулевой строке:
       Console.WriteLine( Array.IndexOf( a[0], 18 ) );
...
```

Листинг 6.5. Работа со ступенчатым массивом с использованием цикла foreach

Массивы объектов

При создании массива, состоящего из элементов ссылочного типа, память выделяется только под ссылки на элементы, а сами элементы необходимо разместить в хипе явным образом. В качестве примера создадим массив из объектов некоторого класса `Monster`:

```
using System;
namespace ConsoleApplication1
{
   class Monster { ... }

   class Class1
   {  static void Main()
      {
         Random rnd = new Random();
         const int n = 5;
         Monster[] stado = new Monster[n];                  // 1
         for ( int i = 0; i < n; ++i )                      // 2
         {
            stado[i] = new Monster( rnd.Next( 1, 100 ),
```

```
                                rnd.Next( 1, 200 ),
                                "Crazy" + i.ToString() );
            }
            foreach ( Monster x in stado ) x.Passport();        // 3
        }
    }
}
```

Результат работы программы:

```
Monster Crazy0   health = 18 ammo = 94
Monster Crazy1   health = 85 ammo = 75
Monster Crazy2   health = 13 ammo = 6
Monster Crazy3   health = 51 ammo = 104
Monster Crazy4   health = 68 ammo = 114
```

Пример поиска в массиве объектов приведен в конце следующей лекции.

Символы и строки

Обработка текстовой информации является, вероятно, одной из самых распространенных задач в современном программировании, и C# предоставляет для ее решения широкий набор средств: отдельные символы, массивы символов, изменяемые и неизменяемые строки и регулярные выражения.

Символы

Символьный тип `char` предназначен для хранения символов в кодировке Unicode. Символьный тип относится к встроенным типам данных C# и соответствует стандартному классу `Char` библиотеки .NET из пространства имен `System`. В этом классе определены статические методы, позволяющие задать вид и категорию символа, а также преобразовать символ в верхний или нижний регистр и в число. Некоторые методы приведены в таблилце 6.2, с остальными можно ознакомиться по учебнику [4].

Таблица 6.2. Некоторые методы класса System.Char

Метод	Описание
`GetNumericValue`	Возвращает числовое значение символа, если он является цифрой, и –1 в противном случае
`IsControl`	Возвращает `true`, если символ является управляющим
`IsDigit`	Возвращает `true`, если символ является десятичной цифрой
`IsLetter`	Возвращает `true`, если символ является буквой
`IsLower`	Возвращает `true`, если символ задан в нижнем регистре
`IsUpper`	Возвращает `true`, если символ записан в верхнем регистре
`IsWhiteSpace`	Возвращает `true`, если символ является пробельным (пробел, перевод строки и возврат каретки)
`Parse`	Преобразует строку в символ (строка должна состоять из одного символа)
`ToLower`	Преобразует символ в нижний регистр
`MaxValue`, *`MinValue`*	Возвращают символы с максимальным и минимальным кодами (эти символы не имеют видимого представления)

В листинге 6.6 продемонстрировано использование этих методов.

```
using System;
namespace ConsoleApplication1
{  class Class1
   {  static void Main()
      {
         try
         {
            char b = 'B', c = '\x63', d = '\u0032';                    // 1
            Console.WriteLine( "{0}  {1}  {2}", b, c, d );
            Console.WriteLine( "{0}  {1}  {2}",
                char.ToLower(b), char.ToUpper(c), char.GetNumericValue(d) );
```

```
            char a;
            do                                          // 2
            {
               Console.Write( "Введите символ: " );
               a = char.Parse( Console.ReadLine() );
               Console.WriteLine( "Введен символ {0}, его код – {1}",
                           a, (int)a );
               if (char.IsLetter(a))      Console.WriteLine("Буква");
               if (char.IsUpper(a))       Console.WriteLine("Верхний рег.");
               if (char.IsLower(a))       Console.WriteLine("Нижний рег.");
               if (char.IsControl(a))     Console.WriteLine("Управляющий");
               if (char.IsNumber(a))      Console.WriteLine("Число");
               if (char.IsPunctuation(a)) Console.WriteLine("Разделитель");
            } while (a != 'q');
         }
         catch
         {
            Console.WriteLine( "Возникло исключение" );
            return;
         }
      }
   }
}
```

Листинг 6.6. Использование методов класса System.Char

Массивы символов

Массив символов, как и массив любого иного типа, построен на основе базового класса `Array`. Применение этих методов позволяет эффективно решать некоторые задачи. Простой пример приведен в листинге 6.7.

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

```
        {
            char[] a = { 'm', 'a', 's', 's', 'i', 'v' };                // 1
            char[] b = "a роза упала на лапу азора".ToCharArray();      // 2

            PrintArray( "Исходный массив a:", a );

            int pos = Array.IndexOf( a, 'm' );
            a[pos] = 'M';
            PrintArray( "Измененный массив a:", a );

            PrintArray( "Исходный массив b:", b );
            Array.Reverse( b );
            PrintArray( "Измененный массив b:", b );
        }

        public static void PrintArray( string header, Array a )
        {
            Console.WriteLine( header );
            foreach ( object x in a ) Console.Write( x );
            Console.WriteLine( "\n" );
        }
    }
}
```

Листинг 6.7. Работа с массивом символов

Результат работы программы:

```
Исходный массив a:
massiv

Измененный массив a:
Massiv

Исходный массив b:
a роза упала на лапу азора

Измененный массив b:
ароза упал ан алапу азор а
```

Строки типа string

Тип `string`, предназначенный для работы со строками символов в кодировке Unicode, является встроенным типом C#. Ему соответствует базовый класс `System.String` библиотеки .NET.

Создать строку можно несколькими способами:

```
string s;                    // инициализация отложена
string t = "qqq";            // инициализация строковым литералом
string u = new string(' ', 20); // конструктор создает строку из 20 пробелов
char[] a = { '0', '0', '0' };   // массив для инициализации строки
string v = new string( a );     // создание из массива символов
```

Для строк определены следующие операции:

- присваивание (=);
- проверка на равенство (==);
- проверка на неравенство (!=);
- обращение по индексу ([]);
- сцепление (конкатенация) строк (+).

Несмотря на то, что строки являются ссылочным типом данных, на равенство и неравенство проверяются не ссылки, а значения строк. Строки равны, если имеют одинаковое количество символов и совпадают посимвольно.

Обращаться к отдельному элементу строки по индексу можно только для получения значения, но не для его изменения. Это связано с тем, что строки типа `string` относятся к так называемым неизменяемым типам данных. Методы, изменяющие содержимое строки, на самом деле создают новую копию строки. Неиспользуемые "старые" копии автоматически удаляются сборщиком мусора.

В классе `System.String` предусмотрено множество методов, полей и свойств, позволяющих выполнять со строками практически любые действия. Некоторые элементы класса приведены в таблице 6.3, с остальными можно ознакомиться по учебнику.

Таблица 6.3. Некоторые элементы класса System.String

Название	Вид	Описание
`Compare`	Статический метод	Сравнение двух строк в лексикографическом (алфавитном) порядке. Разные реализации метода позволяют сравнивать строки и подстроки с учетом и без учета регистра и особенностей национального представления дат и т. д.
`Concat`	Статический метод	Конкатенация строк. Метод допускает сцепление произвольного числа строк
`Copy`	Статический метод	Создание копии строки
`Format`	Статический метод	Форматирование в соответствии с заданными спецификаторами формата (см. далее)
`IndexOf`, `IndexOfAny`, *`LastIndexOf`*, `LastIndexOfAny`	Методы	Определение индексов первого и последнего вхождения заданной подстроки или любого символа из заданного набора
`Insert`	Метод	Вставка подстроки в заданную позицию
`Join`	Статический метод	Слияние массива строк в единую строку. Между элементами массива вставляются разделители (см. далее)
`Length`	Свойство	Длина строки (количество символов)
`Split`	Метод	Разделение строки на элементы, используя заданные разделители. Результаты помещаются в массив строк
`Substring`	Метод	Выделение подстроки, начиная с заданной позиции

`ToCharArray`	Метод	Преобразование строки в массив символов
`ToLower`, `ToUpper`	Методы	Преобразование символов строки к нижнему или верхнему регистру

Пример применения методов приведен в листинге 6.8.

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            string s = "прекрасная королева Изольда";
            Console.WriteLine( s );
            string sub = s.Substring( 3 ).Remove( 12, 2 );              // 1
            Console.WriteLine( sub );

            string[] mas = s.Split(' ');                                // 2
            string joined = string.Join( "! ", mas );
            Console.WriteLine( joined );

            Console.WriteLine( "Введите строку" );
            string x = Console.ReadLine();                              // 3
            Console.WriteLine( "Вы ввели строку " + x );

            double a = 12.234;
            int b = 29;
            Console.WriteLine( " a = {0,6:C}  b = {1,2:X}", a, b );     // 4
            Console.WriteLine( " a = {0,6:0.##}  a = {1,5:0.# ' руб. '}",
                               a, b );                                  // 5
        }
    }
}
```

Листинг 6.8. Работа со строками типа string

Результат работы программы:

```
прекрасная королева Изольда
```

```
красная корова Изольда
прекрасная! королева! Изольда
Введите строку
не хочу!
Вы ввели строку не хочу!
a = 12,23р.  b = 1D
a = 12,23 a=29 руб.
```

Форматирование строк

В операторе 4 из листинга 6.7 неявно применяется метод `Format`, который заменяет все вхождения параметров в фигурных скобках значениями соответствующих переменных из списка вывода. После номера параметра можно задать минимальную ширину поля вывода, а также указать спецификатор формата, который определяет форму представления выводимого значения.

В общем виде параметр задается следующим образом:

```
{n [,m[:спецификатор_формата]]}
```

Здесь `n` — номер параметра. Параметры нумеруются с нуля, нулевой параметр заменяется значением первой переменной из списка вывода, первый параметр — второй переменной, и т. д. Параметр `m` определяет минимальную ширину поля, которое отводится под выводимое значение. Если выводимому числу достаточно меньшего количества позиций, неиспользуемые позиции заполняются пробелами. Если числу требуется больше позиций, параметр игнорируется.

Спецификатор формата, как явствует из его названия, определяет формат вывода значения. Например, спецификатор `C` (Currency) означает, что параметр должен форматироваться как валюта с учетом национальных особенностей представления, а спецификатор `X` (Hexadecimal) задает шестнадцатеричную форму представления выводимого значения.

В операторе 5 используются так называемые пользовательские шаблоны форматирования. Если приглядеться, в них нет ничего сложного: после

двоеточия задается вид выводимого значения посимвольно, причем на месте каждого символа может стоять либо #, либо 0. Если указан знак #, на этом месте будет выведена цифра числа, если она не равна нулю. Если указан 0, будет выведена любая цифра, в том числе и 0.

Строки типа StringBuilder

Возможности, предоставляемые классом `string`, широки, однако требование неизменности его объектов может оказаться неудобным. В этом случае для работы со строками применяется класс `StringBuilder`, определенный в пространстве имен `System.Text` и позволяющий изменять значение своих экземпляров. О нем можно прочитать в учебнике [4].

Вопросы и задания для самостоятельной работы студента

1. Перечислите способы описания массивов.
2. Чем отличается хранение в памяти массивов из величин значимого и ссылочного типов?
3. Является ли размерность частью описания массива?
4. Может ли размерность массива описана переменной (а не константой)?
5. Можно ли изменить размерность массива после выделения памяти под него?
6. Какие виды массивов используются в C#?
7. Что происходит, если количество инициализаторов массива не соответствует заявленной размерности?
8. Что происходит при присваивании массивов?
9. Опишите два-три метода сортировки массивов.
10. Опишите основные методы и свойства класса System.Array
11. Какие ограничения имеет оператор foreach по сравнению с оператором for?
12. Что такое кодировка Unicode?
13. Какие средства работы с отдельными символами предоставляет C#?
14. Опишите основные методы и свойства класса string.

15. Можно ли изменить длину строки после того, как память под нее была выделена?
16. Какое основное ограничение имеет класс string?
17. Какие существуют возможности форматирования строк?
18. Перечислите спецификации формата.
19. Опишите пользовательский формат вещественного числа с двумя ведущими нулями и тремя знаками после запятой.
20. Изучите по справочной системе свойства и методы стандартного класса System.Array.
21. Изучите по справочной системе свойства и методы стандартного класса System.String.
22. Изучите по справочной системе свойства и методы стандартного класса System.Char.
23. Изучите по справочной системе свойства и методы стандартного класса System.Text.StringBuilder.
24. Изучите разделы стандарта C#, касающиеся массивов.
25. Изучите разделы стандарта C#, касающиеся символов.
26. Изучите разделы стандарта C#, касающиеся строк.

Лабораторные работы

Лабораторная работа 5. Одномерные массивы

В одномерном массиве, состоящем из `n` вещественных элементов, вычислить:

- сумму отрицательных элементов массива;
- произведение элементов массива, расположенных между максимальным и минимальным элементами.

Упорядочить элементы массива по возрастанию.

Лабораторная работа 6. Двумерные массивы

Дана целочисленная прямоугольная матрица. Определить:

- количество строк, не содержащих ни одного нулевого элемента;
- максимальное из чисел, встречающихся в заданной матрице более одного раза.

Лабораторная работа 7. Строки

В файле находится текст, состоящий не более чем из 50 предложений. Перед выполнением индивидуального варианта задания необходимо считать содержимое этого файла в массив строк, предусмотрев обработку исключений.

В качестве результата выполнения работы вывести на консоль предложения, преобразованные в соответствии с вариантом задания. Каждое предложение начинать с новой строки.

Задание выполнить двумя способами: без использования элементов стандартных классов System.Array, System.Char и System.String и с их использованием.

Задание: упорядочить предложения по возрастанию количества содержащихся в них слов.

Классы: подробности

Перегрузка методов и операций класса. Рекурсивные методы, методы с переменным числом параметров. Индексаторы. Деструкторы.

Перегрузка методов

Презентацию к данной лекции Вы можете скачать ссылка: здесь - http://old.intuit.ru/department/pl/phlcsharp/7/csharp07.ppt.

Часто бывает удобно, чтобы методы, реализующие один и тот же алгоритм для различных типов данных, имели одно и то же имя. Использование нескольких методов с одним и тем же именем, но различными типами параметров называется перегрузкой методов.

Компилятор определяет, какой именно метод требуется вызвать, по типу фактических параметров. Этот процесс называется разрешением (resolution) перегрузки. Тип возвращаемого методом значения в разрешении не участвует. Механизм разрешения основан на достаточно сложном наборе правил, смысл которых сводится к тому, чтобы использовать метод с наиболее подходящими аргументами и выдать сообщение, если такой не найдется. Допустим, имеется четыре варианта метода, определяющего наибольшее значение:

```
// Возвращает наибольшее из двух целых:
int max( int a, int b )
// Возвращает наибольшее из трех целых:
int max( int a, int b, int c )
// Возвращает наибольшее из первого параметра и длины второго:
int max ( int a, string b )
// Возвращает наибольшее из второго параметра и длины первого:
int max ( string b, int a )
...
Console.WriteLine( max( 1, 2 ) );
Console.WriteLine( max( 1, 2, 3 ) );
Console.WriteLine( max( 1, "2" ) );
Console.WriteLine( max( "1", 2 ) );
```

При вызове метода `max` компилятор выбирает вариант метода,

соответствующий типу передаваемых в метод аргументов (в приведенном примере будут последовательно вызваны все четыре варианта метода).

Если точного соответствия не найдено, выполняются неявные преобразования типов в соответствии с общими правилами. Если преобразование невозможно, выдается сообщение об ошибке. Если соответствие на одном и том же этапе может быть получено более чем одним способом, выбирается вариант, содержащий меньшее количество и длину преобразований. Если существует несколько вариантов, из которых невозможно выбрать лучший, выдается сообщение об ошибке.

Перегруженные методы имеют одно имя, но должны различаться параметрами, точнее их типами и способами передачи (`out` или `ref`). Например, методы, заголовки которых приведены ниже, имеют различные сигнатуры и считаются перегруженными:

```
int max( int a, int b )
int max( int a, ref int b )
```

Перегрузка широко используется в классах библиотеки .NET. Например, в стандартном классе `Console` метод `WriteLine` перегружен 19 раз для вывода величин разных типов. Пример класса с перегруженными методами приведен в конце этой лекции (листинг 7.3).

Рекурсивные методы

Рекурсивным называется метод, который вызывает сам себя. Такая рекурсия называется прямой. Существует еще косвенная рекурсия, когда два или более метода вызывают друг друга. Если метод вызывает себя, в стеке создается копия значений его параметров, как и при вызове обычного метода, после чего управление передается первому исполняемому оператору метода. При повторном вызове этот процесс повторяется.

Ясно, что для завершения вычислений каждый рекурсивный метод должен содержать хотя бы одну нерекурсивную ветвь алгоритма, заканчивающуюся оператором возврата. При завершении метода соответствующая часть стека освобождается, и управление передается

вызывающему методу, выполнение которого продолжается с точки, следующей за рекурсивным вызовом.

Классическим примером рекурсивной функции является функция вычисления факториала (это не означает, что факториал следует вычислять именно так). Для того чтобы получить значение факториала числа `n`, требуется умножить на `n` факториал числа (`n - 1`). Известно также, что `0! = 1` и `1 != 1`.

```
long fact( long n ) {
    if ( n == 0 || n == 1 ) return 1;        // нерекурсивная ветвь
    return ( n * fact( n – 1 ) );            // рекурсивная ветвь
}
```

То же самое можно записать короче:

```
long fact( long n ) {
    return ( n > 1 ) ? n * fact( n – 1 ) : 1;
}
```

Рекурсивные методы чаще всего применяют для компактной реализации рекурсивных алгоритмов, а также для работы со структурами данных, описанными рекурсивно, например, с двоичными деревьями.

К достоинствам рекурсии можно отнести компактность записи, к недостаткам — расход времени и памяти на повторные вызовы метода и передачу ему копий параметров, а главное, опасность переполнения стека.

Методы с переменным количеством аргументов

Иногда бывает удобно создать метод, в который можно передавать разное количество аргументов. Язык C# предоставляет такую возможность с помощью ключевого слова `params`. Параметр, помеченный этим ключевым словом, размещается в списке параметров последним и обозначает массив заданного типа неопределенной длины, например:

```
public int Calculate( int a, out int c, params int[] d ) …
```

В этот метод можно передать три и более параметров. Внутри метода к параметрам, начиная с третьего, обращаются как к обычным элементам массива. Количество элементов массива получают с помощью его свойства `Length`. В качестве примера рассмотрим метод вычисления среднего значения элементов массива (листинг 7.1).

```
using System;
namespace ConsoleApplication1
{    class Class1
   {
      public static double Average( params int[] a )
      {
         if ( a.Length == 0 )
            throw new Exception( "Недостаточно аргументов в методе" );

         double av = 0;
         foreach ( int elem in a ) av += elem;
         return av / a.Length;
      }

      static void Main()
      {   try
         {
            int[] a = { 10, 20, 30 };
            Console.WriteLine( Average( a ) );           // 1
            int[] b = { -11, -4, 12, 14, 32, -1, 28 };
            Console.WriteLine( Average( b ) );           // 2
            short z = 1, e = 12;
            byte v = 100;
            Console.WriteLine( Average( z, e, v ) );     // 3
            Console.WriteLine( Average() );              // 4
         }
         catch( Exception e )
         {
            Console.WriteLine( e.Message );
            return;
         }
```

```
        }
    }
}
```

Листинг 7.1. Метод с переменным числом параметров

Результат работы программы:

```
10
20
40
Недостаточно аргументов в методе
```

Параметр-массив может быть только один и должен располагаться последним в списке. Соответствующие ему аргументы должны иметь типы, для которых возможно неявное преобразование к типу массива.

Метод Main

Метод, которому передается управление после запуска программы, должен иметь имя `Main` и быть статическим. Он может принимать параметры из внешнего окружения и возвращать значение в вызвавшую среду. Предусматривается два варианта метода — с параметрами и без параметров:

```
// без параметров:
static тип  Main() { ... }
static void Main() { ... }
// с параметрами:
static тип  Main( string[] args ) { /* ... */ }
static void Main( string[] args ) { /* ... */ }
```

Параметры, разделяемые пробелами, задаются при запуске программы из командной строки после имени исполняемого файла программы. Они передаются в массив `args`.

Если метод возвращает значение, оно должно быть целого типа, если не возвращает, он должен описываться как `void`. Ненулевое значение обычно означает аварийное завершение. Возвращаемое значение

анализируется в командном файле, из которого запускается программа. Обычно это делается для того, чтобы можно было принять решение, выполнять ли командный файл дальше. В листинге 7.2 приведен пример метода `Main`, который выводит свои аргументы и ожидает нажатия любой клавиши.

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main( string[] args )
        {
            foreach( string arg in args ) Console.WriteLine( arg );
            Console.Read();
        }
    }
}
```

Листинг 7.2. Параметры метода Main

Индексаторы

Индексатор представляет собой разновидность свойства. Если у класса есть скрытое поле, представляющее собой массив, то с помощью индексатора можно обратиться к элементу этого массива, используя имя объекта и номер элемента массива в квадратных скобках. Иными словами, индексатор — это такой "умный" индекс для объектов.

Синтаксис индексатора аналогичен синтаксису свойства:

```
атрибуты спецификаторы тип this [ список_параметров ]
{
    get код_доступа
    set код_доступа
}
```

Индексаторы чаще всего объявляются со спецификатором `public`, поскольку они входят в интерфейс объекта. Атрибуты и спецификаторы могут отсутствовать.

Код доступа представляет собой блоки операторов, которые выполняются при получении (`get`) или установке значения (`set`) элемента массива. Может отсутствовать либо часть `get`, либо `set`, но не обе одновременно. Если отсутствует часть `set`, индексатор доступен только для чтения (read-only), если отсутствует часть `get`, индексатор доступен только для записи (write-only).

Список параметров содержит одно или несколько описаний индексов, по которым выполняется доступ к элементу. Чаще всего используется один индекс целого типа.

Индексаторы в основном применяются для создания специализированных массивов, на работу с которыми накладываются какие-либо ограничения. Примеры работы с индексаторами приведены в учебнике.

Язык C# допускает использование многомерных индексаторов. Они описываются аналогично обычным и применяются в основном для контроля за занесением данных в многомерные массивы и выборке данных из многомерных массивов, оформленных в виде классов.

Операции класса

Класс — это полноценный тип данных, поэтому программист может определить для него собственные операции. Вы уже сталкивались с перегрузкой (т.е. переопределением) операций: например, знак " + " для арифметических типов означает сложение, а для строк — склеивание. Операции перегружают в основном для классов, с помощью которых задают какие-либо математические понятия, т.е. тогда, когда знаки операции имеют общепринятую семантику. Можно задать операцию сложения для класса `Monster`, описанного в листинге 5.8, но будет ли понятен ее смысл пользователям этого класса? Примеры перегрузки приведены именно для этого класса только для того, чтобы не тратить время на изучение нового класса.

Унарные операции

Можно определять в классе следующие унарные операции:

```
+   -   !   ~   ++   --   true   false
```

Синтаксис объявителя унарной операции:

```
тип operator унарная_операция ( параметр )
```

Примеры заголовков унарных операций:

```
public static int operator ++( MyObject m )
public static MyObject operator --( MyObject m )
public static bool operator true( MyObject m )
```

Параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется. Операция должна возвращать:

- для операций `+`, `-`, `!` и `~` величину любого типа;
- для операций `++` и `--` величину типа класса, для которого она определяется;
- для операций `true` и `false` величину типа `bool`.

Операции не должны изменять значение передаваемого им операнда. Операция, возвращающая величину типа класса, для которого она определяется, должна создать новый объект этого класса, выполнить с ним необходимые действия и передать его в качестве результата.

Примечание

Префиксный и постфиксный инкремент не различаются (для них может существовать только одна реализация, которая вызывается в обоих случаях).

Пример перегрузки операции инкремента (примем, что увеличение монстра на единицу должно увеличивать его здоровье):

```
class Monster {
  public static Monster operator ++( Monster m )
  {
    Monster temp = new Monster();
```

```
    temp.health = m.health + 1;
    return temp;
  }
  …
}

…
Monster vasia = new Monster();
++vasia; vasia++;
```

Бинарные операции

Можно определять в классе следующие бинарные операции:

```
+  -  *  /  %  &  |  ^  <<  >>  ==  !=  >  <  >=  <=
```

Синтаксис объявителя бинарной операции:

```
тип operator бинарная_операция (параметр1, параметр2)
```

Примеры заголовков бинарных операций:

```
public static MyObject operator +  ( MyObject m1, MyObject m2 )
public static bool     operator == ( MyObject m1, MyObject m2 )
```

Хотя бы один параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется. Операция может возвращать величину любого типа.

Операции == и !=, > и <, >= и <= определяются только парами и обычно возвращают логическое значение. Чаще всего в классе определяют операции сравнения на равенство и неравенство для того, чтобы обеспечить сравнение объектов, а не их ссылок, как определено по умолчанию для ссылочных типов.

Сложные операции присваивания (например, +=) определять не требуется, да это и невозможно. При выполнении такой операции автоматически вызываются сначала операция сложения, а потом присваивания.

Пример описания бинарных операций для класса `Monster` (сложение с константой):

```
class Monster {
  public static Monster operator +( Monster m, int k )
  { Monster temp = new Monster();
    temp.ammo = m.ammo + k;
    return temp;
  }
 public static Monster operator +( int k, Monster m )
  { Monster temp = new Monster();
    temp.ammo = m.ammo + k;
    return temp;
  }
  …
}
 …
Monster vasia = new Monster();
Monster masha = vasia + 10;
Monster petya = 5 + masha;
…
```

Операции преобразования типа

Операции преобразования типа обеспечивают возможность явного и неявного преобразования между пользовательскими типами данных. Синтаксис объявителя операции преобразования типа:

```
implicit operator тип ( параметр )      // неявное преобразование
explicit operator тип ( параметр )      // явное преобразование
```

Эти операции выполняют преобразование из типа параметра в тип, указанный в заголовке операции. Одним из этих типов должен быть класс, для которого определяется операция. Таким образом, операции выполняют преобразование либо типа класса к другому типу, либо наоборот. Преобразуемые типы не должны быть связаны отношениями наследования. Примеры операций преобразования типа для класса `Monster`, описанного ранее:

```
public static implicit operator int( Monster m )
{
   return m.health;
}
public static explicit operator Monster( int h )
{
   return new Monster( h, 100, "FromInt" );
}
```

Ниже приведены примеры использования этих преобразований в программе. Не надо искать в них смысл, они просто иллюстрируют синтаксис:

```
Monster Masha = new Monster( 200, 200, "Masha" );
int i = Masha;                    // неявное преобразование
Masha = (Monster) 500;            // явное преобразование
```

Неявное преобразование выполняется автоматически:

- при присваивании объекта переменной целевого типа, как в примере;
- при использовании объекта в выражении, содержащем переменные целевого типа;
- при передаче объекта в метод на место параметра целевого типа;
- при явном приведении типа.

Явное преобразование выполняется при использовании операции приведения типа.

Все операции класса должны иметь разные сигнатуры. В отличие от других видов методов, для операций преобразования тип возвращаемого значения включается в сигнатуру, иначе нельзя было бы определять варианты преобразования данного типа в несколько других. Ключевые слова `implicit` и `explicit` в сигнатуру не включаются, следовательно, для одного и того же преобразования нельзя определить одновременно явную и неявную версию.

Неявное преобразование следует определять так, чтобы при его выполнении не возникала потеря точности и не генерировались

исключения. Если эти ситуации возможны, преобразование следует описать как явное.

Пример класса с перегруженными методами и операциями

Рассмотрим решение следующей задачи.

В текстовом файле хранится база отдела кадров предприятия. На предприятии 100 сотрудников. Каждая строка файла содержит запись об одном сотруднике. Формат записи: фамилия и инициалы (30 поз., фамилия должна начинаться с первой позиции), год рождения (5 поз.), оклад (10 поз.). Написать программу, которая по заданной фамилии выводит на экран сведения о сотруднике, подсчитывая средний оклад всех запрошенных сотрудников.

Для решения этой задачи логично создать класс "сотрудник" и организовать из экземпляров этого класса массив. При описании класса полезно задаться следующим вопросом: какие обязанности должны быть возложены на этот класс? Очевидно, что первая обязанность — хранение сведений о сотруднике. Чтобы воспользоваться этими сведениями, клиент (то есть код, использующий класс) должен иметь возможность получить эти сведения, изменить их и вывести на экран. Кроме этого, для поиска сотрудника желательно иметь возможность сравнивать его имя с заданным.

Поля класса сделаем в соответствии с принципом инкапсуляции закрытыми, а доступ к ним организуем с помощью свойств. Это позволит контролировать процесс занесения данных в поля. Для идентификации попытки ввода неверных данных воспользуемся механизмом исключений. Чтобы сделать класс более универсальным, введем в него возможность увеличивать и уменьшать оклад сотрудника с помощью перегруженных операций класса.

На этом примере мы формализуем общий порядок создания программы, которого полезно придерживаться при решении даже простейших задач

I. Исходные данные, результаты и промежуточные величины.

Исходные данные. База сотрудников находится в текстовом файле. Прежде всего надо решить, хранить ли в оперативной памяти одновременно всю информацию из файла или можно обойтись буфером на одну строку. Если бы сведения о сотруднике запрашивались однократно, можно было бы остановиться на втором варианте, но поскольку поиск по базе будет выполняться более одного раза, всю информацию желательно хранить в оперативной памяти, поскольку многократное чтение из файла крайне нерационально.

Максимальное количество строк файла по условию задачи ограничено, поэтому можно выделить для их хранения массив из 100 элементов. Каждый элемент массива будет содержать сведения об одном сотруднике, организованные в виде класса.

Примечание

Строго говоря, для решения этой конкретной задачи запись о сотруднике может быть просто строкой, из которой при необходимости выделяется подстрока с окладом, преобразуемая затем в число, но мы для общности и удобства дальнейшей модификации программы и демонстрации синтаксиса будем использовать класс.

В программу по условию требуется также вводить фамилии искомых сотрудников. Очередную вводимую фамилию будем также хранить в стандартной строке типа `string`.

Результаты. В результате работы программы требуется вывести на экран требуемые элементы исходного массива. Поскольку эти результаты представляют собой выборку из исходных данных, дополнительная память для них не отводится. Кроме того, необходимо подсчитать средний оклад для найденных сотрудников. Для этого необходима переменная вещественного типа.

Промежуточные величины. Для поиска среднего оклада необходимо подсчитать количество сотрудников, для которых выводились сведения. Заведем для этого переменную целого типа.

II. Алгоритм решения задачи очевиден:

1. Ввести из файла в массив сведения о сотрудниках.

2. Организовать цикл вывода сведений о сотруднике:
   - ввести с клавиатуры фамилию;
   - выполнить поиск сотрудника в массиве;
   - увеличить суммарный оклад и счетчик количества сотрудников;
   - вывести сведения о сотруднике или сообщение об их отсутствии;
3. Вывести средний оклад.

Необходимо решить, каким образом будет производиться выход из цикла вывода сведений о сотрудниках. Условимся, что для выхода из цикла вместо фамилии следует просто нажать клавишу Enter. Текст программы приведен в листинге 7.3.

```
using System;
using System.IO;
namespace ConsoleApplication1
{
   class Man {                          // 1
      const int l_name = 30;
      string name;
      int birth_year;
      double pay;
      public Man()           // конструктор по умолчанию
      {
         name = "Anonymous";
         birth_year = 0;
         pay = 0;
      }
      public Man(string s)                                  // 2
      {
         name       = s.Substring(0, l_name);
         birth_year  = Convert.ToInt32(s.Substring(l_name, 4));
         pay        = Convert.ToDouble(s.Substring(l_name + 4));
         if (birth_year < 0) throw new FormatException();
         if (pay < 0)        throw new FormatException();
      }
      public override string ToString()                     // 3
      {
```

```
        return string.Format( "Name: {0,30} birth: {1} pay: {2:F2}",
                              name, birth_year, pay);
    }
    public int Compare(string name) // сравнение фамилии
    {
        return (string.Compare(this.name, 0,
                               name + " ", 0, name.Length + 1,
                               StringComparison.OrdinalIgnoreCase));
    }
    // --------------------  свойства класса -------------------------
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
    public int Birth_year
    {
        get { return birth_year; }
        set { if (value > 0) birth_year = value;
              else throw new FormatException(); }
    }
    public double Pay
    {
        get { return pay; }
        set { if (value > 0) pay = value;
              else throw new FormatException(); }
    }
          // ------------------  операции класса ------------------------------
    public static double operator +(Man man1, double a)
    {
        man1.pay += a;
        return man1.pay;
    }
    public static double operator +(double a, Man man1)
    {
        man1.pay += a;
        return man1.pay;
    }
    public static double operator -(Man man1, double a)
```

```
        {
            man1.pay -= a;
            if ( man1.pay < 0 ) throw new FormatException();
            return man1.pay;
        }
    };

    class Class1
    {
        static void Main()
        {
            Man[] dbase = new Man[100];
            int n = 0;
            try
            {
                StreamReader f = new StreamReader("dbase.txt");          // 4

                string s;
                int i = 0;

                while ((s = f.ReadLine()) != null)                       // 5
                {
                    dbase[i] = new Man(s);
                    Console.WriteLine(dbase[i]);
                    ++i;
                }
                n = i - 1;
                f.Close();
            }
            catch (FileNotFoundException e)
            {
                Console.WriteLine(e.Message);
                Console.WriteLine(" Проверьте правильность имени файла!");
                return;
            }

            catch (IndexOutOfRangeException)
            {
                Console.WriteLine("Слишком большой файл!");
```

```
            return;
        }
        catch (FormatException)
        {
            Console.WriteLine("Недопустимая дата рождения или оклад");
            return;
        }

        catch (Exception e)
        {
            Console.WriteLine("Ошибка: " + e.Message);
            return;
        }

        int n_man = 0;
        double mean_pay = 0;
        Console.WriteLine("Введите фамилию сотрудника");
        string name;
        while ((name = Console.ReadLine()) != "")                    // 6
        {
            bool not_found = true;
            for ( int k = 0; k <= n; ++k )
            {
                Man man = dbase[k];
                if (man.Compare(name) == 0)
                {
                    Console.WriteLine(man);
                    ++n_man; mean_pay += man.Pay;
                                not_found = false;
                }
            }
            if (not_found) Console.WriteLine("Такого сотрудника нет");
            Console.WriteLine(
    "Введите фамилию сотрудника или Enter для окончания");
        }
        if (n_man > 0)
                Console.WriteLine("Средний оклад: {0:F2}", mean_pay / n_m
    }
}
```

```
}
```

Листинг 7.3. Поиск в массиве классов

В операторе 1 описан класс `Man` для хранения информации об одном сотруднике. Кроме полей данных, в ней задан конструктор, выполняющий заполнение полей (оператор 2), переопределен метод преобразования в строку (оператор 3), что позволяет выводить экземпляр объекта на экран с помощью метода `WriteLine` класса `Console`, а также описан метод `Compare`, в котором фамилия сотрудника сравнивается с заданной через параметр.

Сравнение фамилии выполняется с помощью одного из вариантов метода `Compare` класса `string`, который позволяет сравнивать подстроки без учета регистра. После заданной фамилии добавлен пробел, поскольку если пробела нет, то искомая фамилия является частью другой, и эта строка нам не подходит.

Свойства и операции класса позволяют выполнять ввод и корректировку оклада и года рождения.

Входной файл следует создать до первого запуска программы в соответствии с форматом, заданным в условии задачи. Можно использовать любой текстовый редактор, поддерживающий кодировку Unicode (например, Блокнот). Файл для целей тестирования должен состоять из нескольких строк, причем необходимо предусмотреть случай, когда одна фамилия является частью другой (например, Иванов и Ивановский). Не забудьте проверить, выдается ли диагностическое сообщение, если файл не найден.

В операторе 4 описан экземпляр стандартного класса символьного потока. Цикл 5 выполняет построчное считывание из файла в строку `s` и заполнение очередного элемента массива `dbase`. В операторе 6 организуется цикл просмотра массива. Просматриваются только заполненные при вводе элементы.

Деструкторы

В C# существует специальный вид метода, называемый деструктором.

Он вызывается сборщиком мусора непосредственно перед удалением объекта из памяти. В деструкторе описываются действия, гарантирующие корректность последующего удаления объекта, например, проверяется, все ли ресурсы, используемые объектом, освобождены (файлы закрыты, удаленное соединение разорвано и т. п.)

Синтаксис деструктора:

```
[ атрибуты ] [ extern ] ~имя_класса()
тело
```

Как видно из определения, деструктор не имеет параметров, не возвращает значения и не требует указания спецификаторов доступа. Его имя совпадает с именем класса и предваряется тильдой (`~`), символизирующей обратные по отношению к конструктору действия. Тело деструктора представляет собой блок или просто точку с запятой, если деструктор определен как внешний (`extern`).

Сборщик мусора удаляет объекты, на которые нет ссылок. Он работает в соответствии со своей внутренней стратегией в неизвестные для программиста моменты времени. Поскольку деструктор вызывается сборщиком мусора, невозможно гарантировать, что деструктор будет обязательно вызван в процессе работы программы. Следовательно, его лучше использовать только для гарантии освобождения ресурсов, а "штатное" освобождение выполнять в другом месте программы.

Вложенные типы

В классе можно определять типы данных, внутренние по отношению к классу. Так определяются вспомогательные типы, которые используются только содержащим их классом. Механизм вложенных типов позволяет скрыть ненужные детали и более полно реализовать принцип инкапсуляции. Непосредственный доступ извне к такому классу невозможен (имеется в виду доступ по имени без уточнения). Для вложенных типов можно использовать те же спецификаторы, что и для полей класса.

Например, введем в наш класс `Monster` вспомогательный класс `Gun`. Объекты этого класса без "хозяина" бесполезны, поэтому его можно

определить как внутренний:

```
using System;
namespace ConsoleApplication1
{   class Monster
    {
        class Gun
        {
            ...
        }
        ...
    }
}
```

Помимо классов, вложенными могут быть и другие типы данных: интерфейсы, структуры и перечисления. Мы рассмотрим их позже.

Вопросы и задания для самостоятельной работы студента

1. Что такое индексатор, для каких классов он применяется?
2. Может ли статический конструктор инициализировать поля экземпляра?
3. Зачем нужна перегрузка методов и операций?
4. Опишите способы перегрузки операций.
5. Для каких классов имеет смысл использовать перегруженные операции?
6. Какой тип имеет параметр унарной операции класса?
7. Можно ли перегрузить операции простого и сложного присваивания?
8. Как передать программе параметры из внешнего окружения?

Лабораторная работа 8. Классы и операции

Теоретический материал: глава 7.

Каждый разрабатываемый класс должен, как правило, содержать следующие элементы: скрытые поля, конструкторы с параметрами и без

параметров, методы; свойства, индексаторы; перегруженные операции. Функциональные элементы класса должны обеспечивать непротиворечивый, полный, минимальный и удобный интерфейс класса. При возникновении ошибок должны выбрасываться исключения.

Задание

Описать класс для работы с одномерным массивом целых чисел (вектором). Обеспечить следующие возможности:

- задание произвольных целых границ индексов при создании объекта;
- обращение к отдельному элементу массива с контролем выхода за пределы массива;
- выполнение операций поэлементного сложения и вычитания массивов с одинаковыми границами индексов;
- выполнение операций умножения и деления всех элементов массива на скаляр;
- вывода на экран элемента массива по заданному индексу и всего массива.

Написать программу, демонстрирующую все разработанные элементы класса.

Наследование классов

Организация иерархий классов. Раннее и позднее связывание. Виртуальные методы. Абстрактные и бесплодные классы. Виды взаимоотношений между классами.

Презентацию к данной лекции Вы можете скачать ссылка: здесь - http://old.intuit.ru/department/pl/phlcsharp/8/csharp08.ppt.

Управлять большим количеством разрозненных классов достаточно сложно. С этой проблемой можно справиться путем упорядочивания и ранжирования классов, то есть объединяя общие для нескольких классов свойства в одном классе и используя его в качестве базового.

Эту возможность предоставляет механизм наследования, который является мощнейшим инструментом ООП. Он позволяет строить иерархии, в которых классы-потомки получают свойства классов-предков и могут дополнять их или изменять. Таким образом, наследование обеспечивает важную возможность многократного использования кода.

Классы, расположенные ближе к началу иерархии, объединяют в себе общие черты для всех нижележащих классов. По мере продвижения вниз по иерархии классы приобретают все больше конкретных особенностей.

Описание класса-потомка

Класс в C# может иметь произвольное количество потомков и только одного предка. При описании класса имя его предка записывается в заголовке класса после двоеточия. Если имя предка не указано, предком считается базовый класс всей иерархии `System.Object`:

```
[ атрибуты ] [ спецификаторы ] class имя_класса [ : предки ]
    тело класса
```

Примечание

Обратите внимание, что слово "предки" присутствует в описании

класса во множественном числе, хотя класс может иметь только одного предка. Причина в том, что класс наряду с единственным предком может наследовать от интерфейсов — специального вида классов, не имеющих реализации. Интерфейсы рассматриваются на следующей лекции.

Рассмотрим наследование классов на примере. Ранее был описан класс `Monster`, моделирующий персонаж компьютерной игры. Допустим, нам требуется ввести в игру еще один тип персонажей, который должен обладать свойствами объекта `Monster`, а кроме того, уметь думать. Будет логично сделать новый объект потомком объекта `Monster` (листинг 8.11).

```
using System;
namespace ConsoleApplication1
{
    class Monster
    {
        ...
    }

    class Daemon : Monster
    {
        public Daemon()
        {
            brain = 1;
        }

        public Daemon( string name, int brain ) : base( name )          // 1
        {
            this.brain = brain;
        }

        public Daemon( int health, int ammo, string name, int brain )
            : base( health, ammo, name )                                // 2
        {
            this.brain = brain;
        }
```

```
        new public void Passport()                          // 3
        {
            Console.WriteLine(
                "Daemon {0} \t health = {1} ammo = {2} brain = {3}",
                Name, Health, Ammo, brain );
        }

        public void Think()                                 // 4
        {
            Console.Write( Name + " is" );
            for ( int i = 0; i < brain; ++i ) Console.Write( " thinking" );
            Console.WriteLine( "..." );
        }

        int brain;        // закрытое поле
    }

    class Class1
    {   static void Main()
        {
            Daemon Dima = new Daemon( "Dima", 3 );                  // 5
            Dima.Passport();                                        // 6
            Dima.Think();                                           // 7
            Dima.Health -= 10;                                      // 8
            Dima.Passport();
        }
    }
}
```

Листинг 8.1. Класс Daemon, потомок класса Monster

В классе `Daemon` введены закрытое поле `brain` и метод `Think`, определены собственные конструкторы, а также переопределен метод `Passport`. Все поля и свойства класса `Monster` наследуются в классе `Daemon`.

Результат работы программы:

```
Daemon Dima      health = 100 ammo = 100 brain = 3
```

```
Dima is thinking thinking thinking...
Daemon Dima      health = 90 ammo = 100 brain = 3
```

Как видите, экземпляр класса `Daemon` с одинаковой легкостью использует как собственные (операторы 5–7), так и унаследованные (оператор 8) элементы класса. Рассмотрим общие правила наследования.

Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы. Порядок вызова конструкторов определяется приведенными ниже правилами.

- Если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса без параметров.
- Для иерархии, состоящей из нескольких уровней, конструкторы базовых классов вызываются, начиная с самого верхнего уровня. После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса. Таким образом, каждый конструктор инициализирует свою часть объекта.
- Если конструктор базового класса требует указания параметров, он должен быть явным образом вызван в конструкторе производного класса в списке инициализации. Вызов выполняется с помощью ключевого слова `base`. Вызывается та версия конструктора, список параметров которой соответствует списку аргументов, указанных после слова `base`.

Поля, методы и свойства класса наследуются, поэтому при желании заменить элемент базового класса новым элементом следует явным образом указать компилятору свое намерение с помощью ключевого слова `new`. В листинге 8.1 таким образом переопределен метод вывода информации об объекте `Passport`. Метод `Passport` класса `Daemon` замещает соответствующий метод базового класса, однако возможность доступа к методу базового класса из метода производного класса сохраняется. Для этого перед вызовом метода указывается все то же волшебное слово `base`, например:

```
base.Passport();
```

Элементы базового класса, определенные как `private`, в производном классе недоступны. Поэтому в методе `Passport` для доступа к полям `name`, `health` и `ammo` пришлось использовать соответствующие свойства базового класса. Другое решение заключается в том, чтобы определить эти поля со спецификатором `protected`, в этом случае они будут доступны методам всех классов, производных от `Monster`. Оба решения имеют свои достоинства и недостатки.

Во время выполнения программы объекты хранятся в отдельных переменных, массивах или других коллекциях. Во многих случаях удобно оперировать объектами одной иерархии единообразно, то есть использовать один и тот же программный код для работы с экземплярами разных классов. Это возможно благодаря тому, что объекту базового класса можно присвоить объект производного класса.

Попробуем описать массив объектов базового класса и занести туда объекты производного класса. В листинге 8.2 в массиве типа `Monster` хранятся два объекта типа `Monster` и один — типа `Daemon`.

```
using System;
namespace ConsoleApplication1
{
   class Monster
   {
      ...
   }

   class Daemon : Monster
   {
      ... //
   }

   class Class1
   {   static void Main()
      {
         const int n = 3;
         Monster[] stado = new Monster[n];

         stado[0] = new Monster( "Monia" );
```

```
            stado[1] = new Monster( "Monk" );
            stado[2] = new Daemon ( "Dimon", 3 );

            foreach ( Monster elem in stado ) elem.Passport();          // 1

            for ( int i = 0; i < n; ++i ) stado[i].Ammo = 0;            // 2
            Console.WriteLine();

            foreach ( Monster elem in stado ) elem.Passport();          // 3
        }
    }
}
```

Листинг 8.2. Массив объектов разных типов

Результат работы программы:

```
Monster Monia    health = 100 ammo = 100
Monster Monk     health = 100 ammo = 100
Monster Dimon    health = 100 ammo = 100

Monster Monia    health = 100 ammo = 0
Monster Monk     health = 100 ammo = 0
Monster Dimon    health = 100 ammo = 0
```

Результат радует нас только частично: объект типа `Daemon` действительно можно поместить в массив, состоящий из элементов типа `Monster`, но для него вызываются только методы и свойства, унаследованные от предка. Это устраивает нас в операторе 2, а в операторах 1 и 3 хотелось бы, чтобы вызывался метод `Passport`, переопределенный в потомке.

Итак, присваивать объекту базового класса объект производного класса можно, но вызываются для него только методы и свойства, определенные в базовом классе. Иными словами, возможность доступа к элементам класса определяется типом ссылки, а не типом объекта, на который она указывает.

Это и понятно: ведь компилятор должен еще до выполнения программы решить, какой метод вызывать, и вставить в код фрагмент, передающий

управление на этот метод (этот процесс называется ранним связыванием). При этом компилятор может руководствоваться только типом переменной, для которой вызывается метод или свойство (например, `stado[i].Ammo`). То, что в этой переменной в разные моменты времени могут находиться ссылки на объекты разных типов, компилятор учесть не может.

Следовательно, если мы хотим, чтобы вызываемые методы соответствовали типу объекта, необходимо отложить процесс связывания до этапа выполнения программы, а точнее — до момента вызова метода, когда уже точно известно, на объект какого типа указывает ссылка. Такой механизм в C# есть — он называется поздним связыванием и реализуется с помощью так называемых виртуальных методов, которые мы незамедлительно и рассмотрим.

Виртуальные методы

При раннем связывании программа, готовая для выполнения, представляет собой структуру, логика выполнения которой жестко определена. Если же требуется, чтобы решение о том, какой из одноименных методов разных объектов иерархии использовать, принималось в зависимости от конкретного объекта, для которого выполняется вызов, то заранее жестко связывать эти методы с остальной частью кода нельзя.

Следовательно, надо каким-то образом дать знать компилятору, что эти методы будут обрабатываться по-другому. Для этого в C# существует ключевое слово `virtual`. Оно записывается в заголовке метода базового класса, например:

```
virtual public void Passport() ...
```

Объявление метода виртуальным означает, что все ссылки на этот метод будут разрешаться в момент его вызова во время выполнения программы. Этот механизм называется поздним связыванием.

Для его реализации необходимо, чтобы адреса виртуальных методов хранились там, где ими можно будет в любой момент воспользоваться, поэтому компилятор формирует для этих методов таблицу виртуальных

методов (Virtual Method Table, VMT). В нее записываются адреса виртуальных методов (в том числе унаследованных) в порядке описания в классе. Для каждого класса создается одна таблица.

Каждый объект во время выполнения должен иметь доступ к VMT. Связь экземпляра объекта с VMT устанавливается с помощью специального кода, автоматически помещаемого компилятором в конструктор объекта.

Если в производном классе требуется переопределить виртуальный метод, используется ключевое слово `override`, например:

```
override public void Passport() ...
```

Переопределенный виртуальный метод должен обладать таким же набором параметров, как и одноименный метод базового класса.

Добавим в листинге 8.2 два волшебных слова — `virtual` и `override` — в описания методов `Passport` соответственно базового и производного классов (листинг 8.3).

```
using System;
namespace ConsoleApplication1
{
    class Monster
    {
        ...
        virtual public void Passport()
        {
            Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",
                        name, health, ammo );
        }
        ...
    }

    class Daemon : Monster
    {
        ...
        override public void Passport()
        {
```

```
            Console.WriteLine(
               "Daemon {0} \t health = {1} ammo = {2} brain = {3}",
               Name, Health, Ammo, brain );
         }
         ...
      }

      class Class1
      {   static void Main()
         {
            const int n = 3;
            Monster[] stado = new Monster[n];

            stado[0] = new Monster( "Monia" );
            stado[1] = new Monster( "Monk" );
            stado[2] = new Daemon ( "Dimon", 3 );

            foreach ( Monster elem in stado ) elem.Passport();

            for ( int i = 0; i < n; ++i ) stado[i].Ammo = 0;
            Console.WriteLine();

            foreach ( Monster elem in stado ) elem.Passport();
         }
      }
   }
```

Листинг 8.3. Виртуальные методы

Результат работы программы:

```
Monster Monia    health = 100 ammo = 100
Monster Monk     health = 100 ammo = 100
Daemon Dimon     health = 100 ammo = 100 brain = 3

Monster Monia    health = 100 ammo = 0
Monster Monk     health = 100 ammo = 0
Daemon Dimon     health = 100 ammo = 0 brain = 3
```

Теперь в циклах 1 и 3 вызывается метод `Passport`, соответствующий

типу объекта, помещенного в массив.

Виртуальные методы базового класса определяют интерфейс всей иерархии. Этот интерфейс может расширяться в потомках за счет добавления новых виртуальных методов. Переопределять виртуальный метод в каждом из потомков не обязательно: если он выполняет устраивающие потомка действия, метод наследуется.

С помощью виртуальных методов реализуется один из основных принципов объектно-ориентированного программирования — полиморфизм. Это слово в переводе с греческого означает "много форм", что в данном случае означает "один вызов — много методов".

Виртуальные методы незаменимы и при передаче объектов в методы в качестве параметров. В параметрах метода описывается объект базового типа, а при вызове в нее передается объект производного класса. Виртуальные методы, вызываемые для объекта из метода, будут соответствовать типу аргумента, а не параметра.

Абстрактные классы

При создании иерархии объектов для исключения повторяющегося кода часто бывает логично выделить их общие свойства в один родительский класс. При этом может оказаться, что создавать экземпляры такого класса не имеет смысла, потому что никакие реальные объекты им не соответствуют. Такие классы называют абстрактными.

Абстрактный класс служит только для порождения потомков. Как правило, в нем задается набор методов, которые каждый из потомков будет реализовывать по-своему. Абстрактные классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах.

Абстрактный класс задает интерфейс для всей иерархии, при этом методам класса может не соответствовать никаких конкретных действий. В этом случае методы имеют пустое тело и объявляются со спецификатором `abstract`.

Если в классе есть хотя бы один абстрактный метод, весь класс также должен быть описан как абстрактный, например:

```
abstract class Spirit
{
   public abstract void Passport();
}
class Monster : Spirit
{
   ...
   override public void Passport()
   {
      Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",
                  name, health, ammo );
   }
   ...
}

class Daemon : Monster
{
   ...
   override public void Passport()
   {
      Console.WriteLine(
         "Daemon {0} \t health = {1} ammo = {2} brain = {3}",
         Name, Health, Ammo, brain );
   }
   ... // полный текст этих классов приведен в главе 12
}
```

Абстрактные классы используются при работе со структурами данных, предназначенными для хранения объектов одной иерархии, и в качестве параметров методов.

Бесплодные классы

В C# есть ключевое слово `sealed`, позволяющее описать класс, от которого, в противоположность абстрактному, наследовать запрещается:

```
sealed class Spirit
    {
        ...
    }
// class Monster : Spirit { ... }          ошибка!
```

Большинство встроенных типов данных описано как `sealed`. Если необходимо использовать функциональность бесплодного класса, применяется не наследование, а вложение, или включение: в классе описывается поле соответствующего типа.

Вложение классов, когда один класс включает в себя поля, являющиеся классами, является альтернативой наследованию при проектировании. Например, если есть объект "двигатель", а требуется описать объект "самолет", логично сделать двигатель полем этого объекта, а не его предком.

Виды взаимоотношений между классами

Механизм наследования классов предоставляет программисту богатейшие возможности организации кода и его многократного использования. Выбор наиболее подходящих средств для целей конкретного проекта основывается на знании механизма их работы и взаимодействия.

Тимоти Бадд [1] приводит интересную классификацию форм наследования. Форма наследования определяет, с какой целью оно используется. Бадд считает, что порождение дочернего класса может быть выполнено по следующим причинам.

- Специализация. Класс-наследник является специализированной формой родительского класса — в наследнике просто переопределяются методы.
- Спецификация. Дочерний класс реализует поведение, описанное в родительском классе. В C# эта форма реализуется наследованием от абстрактного класса.
- Конструирование. Класс-наследник использует методы базового класса, но не является его подтипом.
- Расширение. В класс-потомок добавляют новые методы, расширяя

поведение родительского класса.

- Обобщение. Дочерний класс обобщает поведение базового класса. Обычно такое наследование используется в тех случаях, когда изменить поведение базового класса невозможно (например, базовый класс является библиотечным классом).
- Ограничение. Класс-наследник ограничивает поведение родительского класса.
- Варьирование. Базовый класс и класс-потомок являются вариациями на одну тему, однако связь "класс-подкласс" произвольна, например, "квадрат-прямоугольник" или "прямоугольник-квадрат". Эта форма фактически не отличается от "конструирования", так как класс-наследник, очевидно, "использует методы базового класса, но не является его подтипом".
- Комбинирование. Дочерний класс наследует черты нескольких классов — это множественное наследование (в C# не используется, поскольку множественное наследование запрещено, а наследование от нескольких интерфейсов имеет иной смысл).

Альтернативой наследованию при проектировании классов является вложение, когда один класс включает в себя поля, являющиеся классами. Например, если есть класс "двигатель", а требуется описать класс "самолет", логично сделать двигатель полем этого класса, а не его предком. Вложение представляет отношения классов "Y содержит X" или "Y реализуется посредством X" и обычно реализуется с помощью модели "включение-делегирование", которая иллюстрируется в листинге 8.4.

```
using System;
namespace ConsoleApplication1
{
   class Двигатель
   {   public void Запуск()
       {
          Console.WriteLine( "вжжжж!!" );
       }
   }
   class Самолет
   {  public Самолет()
```

```
        {
            левый  = new Двигатель();
            правый = new Двигатель();
        }
        public void Запустить_двигатели()
        {
            левый.Запуск();
            правый.Запуск();
        }
        Двигатель левый, правый;
    }

    class Class1
    {   static void Main()
        {
        Самолет АН24_1 = new Самолет();
        АН24_1.Запустить_двигатели();
        }
    }
}
```

Листинг 8.4. Модель включения-делегирования

Результат работы программы:

```
вжжжж!!
вжжжж!!
```

В методе `Запустить_двигатели` запрос на запуск двигателей передается, или, как принято говорить, делегируется вложенному классу.

В процессе проектирования объектно-ориентированных программ для описания различного рода взаимоотношений классов и объектов часто используется UML.

UML — Unified Modeling Language — является языком для специфицирования, визуализации, конструирования и документирования программных продуктов, а также используется в бизнес-моделировании и моделировании любых иных (не

программных) систем.

UML позволяет задавать различные аспекты представления системы, например, поведение системы с точки зрения ее внешних пользователей, внутреннюю структуру и различные виды взаимодействия элементов системы. Взаимоотношения классов отображают на диаграмме классов.

Отношение наследования изображается на диаграмме классов в виде незакрашенного треугольника, направленного к базовому классу (предку). Пример изображения наследования с помощью диаграммы классов приведен на рис. 8.1.

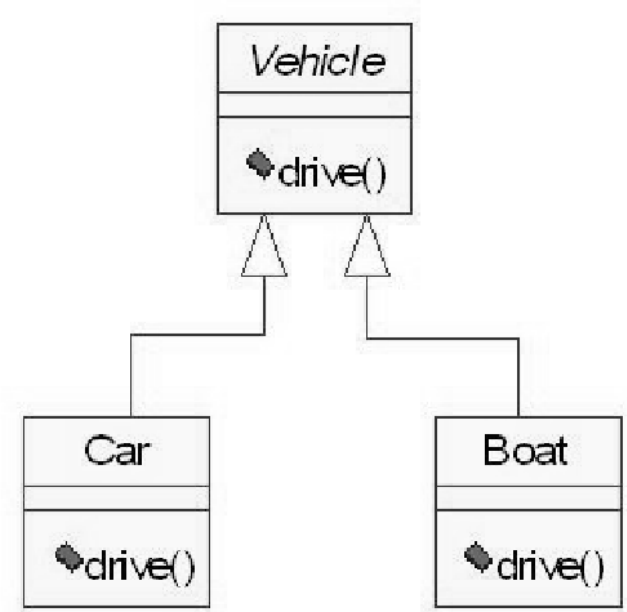


Рис. 8.1. Пример диаграммы UML для наследования

Вложение классов изображается в виде закрашенного или незакрашенного ромба, направленного к владельцу части. Другой конец линии может быть снабжен цифрой, указывающей количество вложенных частей, или звездочкой, если таких частей может быть произвольное количество. Примеры приведены на рис. 8.2.

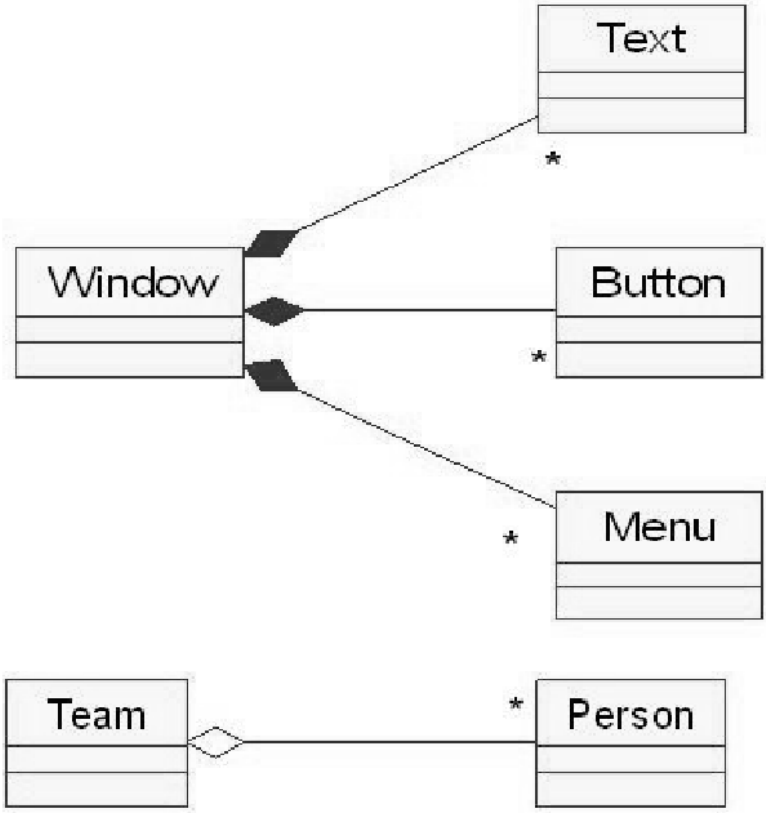


Рис. 8.2. Примеры диаграмм UML для вложения классов

При проектировании классов следует выбирать модель, наиболее точно отражающую смысл взаимоотношений классов, например, моделируемых объектов предметной области.

Класс object

Корневой класс `System.Object` всей иерархии объектов .NET, называемый в C# `object`, обеспечивает всех наследников несколькими важными методами. Производные классы могут использовать эти методы непосредственно или переопределять их.

Класс `object` часто используется и непосредственно при описании типа параметров методов для придания им общности, а также для хранения ссылок на объекты различного типа — таким образом

реализуется полиморфизм.

Открытые методы класса `System.Object` перечислены ниже.

- Метод `Equals` с одним параметром возвращает значение `true`, если параметр и вызывающий объект ссылаются на одну и ту же область памяти. Синтаксис:

```
public virtual bool Equals(object obj);
```

- Метод `Equals` с двумя параметрами возвращает значение `true`, если оба параметра ссылаются на одну и ту же область памяти. Синтаксис:

```
public static bool Equals(object ob1, object ob2);
```

- Метод `GetHashCode` формирует хэш-код объекта и возвращает число, однозначно идентифицирующее объект. Это число используется в различных структурах и алгоритмах библиотеки. Если переопределяется метод `Equals`, необходимо перегрузить и метод `GetHashCode`. Синтаксис:

```
public virtual int GetHashCode();
```

- Метод `GetType` возвращает текущий полиморфный тип объекта, то есть не тип ссылки, а тип объекта, на который она в данный момент указывает. Возвращаемое значение имеет тип `Type`. Это абстрактный базовый класс иерархии, использующийся для получения информации о типах во время выполнения. Синтаксис:

```
public Type GetType();
```

- Метод `ReferenceEquals` возвращает значение `true`, если оба параметра ссылаются на одну и ту же область памяти. Синтаксис:

```
public static bool ReferenceEquals(object ob1, object ob2);
```

- Метод `ToString` по умолчанию возвращает для ссылочных типов полное имя класса в виде строки, а для значимых — значение величины, преобразованное в строку. Этот метод

переопределяют для того, чтобы можно было выводить информацию о состоянии объекта. Синтаксис:

```
public virtual string ToString()
```

В производных объектах эти методы часто переопределяют. Например, можно переопределить метод `Equals` для того, чтобы задать собственные критерии сравнения объектов.

Пример применения и переопределения методов класса `object` для класса `Monster` приведен в листинге 8.5.

```
using System;
namespace ConsoleApplication1
{
   class Monster
   {
      public Monster( int health, int ammo, string name )
      {
         this.health = health;
         this.ammo   = ammo;
         this.name   = name;
      }

      public override bool Equals( object obj )
      {
         if ( obj == null || GetType() != obj.GetType() ) return false;

         Monster temp = (Monster) obj;
         return  health == temp.health &&
                 ammo   == temp.ammo   &&
                 name   == temp.name;
      }

      public override int GetHashCode()
      {
         return name.GetHashCode();
      }
```

```
        public override string ToString()
        {
            return string.Format( "Monster {0} \t health = {1} ammo = {2}",
                    name, health, ammo );
        }

        string name;
        int health, ammo;
    }

    class Class1
    {   static void Main()
        {
            Monster X = new Monster( 80, 80, "Вася" );
            Monster Y = new Monster( 80, 80, "Вася" );
            Monster Z = X;

            if ( X == Y ) Console.WriteLine(" X == Y ");
            else          Console.WriteLine(" X != Y ");

            if ( X == Z ) Console.WriteLine(" X == Z ");
            else          Console.WriteLine(" X != Z ");

            if ( X.Equals(Y) ) Console.WriteLine( " X Equals Y " );
            else               Console.WriteLine( " X not Equals Y " );

            Console.WriteLine(X.GetType());
        }
    }
}
```

Листинг 8.5. Перегрузка методов класса object

Результат работы программы:

```
X != Y
X == Z
X Equals Y
ConsoleApplication1.Monster
```

Анализируя результат работы программы, можно увидеть, что в операции сравнения на равенство сравниваются ссылки, а в перегруженном методе `Equals` — значения. Для концептуального единства можно переопределить и операции отношения.

Вопросы и задания для самостоятельной работы студента

1. Для чего используется наследование?
2. Опишите синтаксис производного класса. Какие спецификаторы доступа применяются в иерархиях?
3. Как вызвать метод базового класса из производного?
4. Опишите порядок вызова конструкторов базовых классов при работе конструктора производного класса.
5. Какие ключевые слова используется при переопределении методов базового класса в производном?
6. Опишите механизмы раннего и позднего связывания.
7. Опишите процесс вызова виртуального метода. Для чего применяются виртуальные методы?
8. Все ли методы следует описывать как виртуальные?
9. Для чего используются абстрактные классы?
10. Какие методы и операции класса object часто перегружают в его потомках?
11. Назовите альтернативы наследованию классов.
12. Изучите по справочной системе свойства и методы класса object.
13. Изучите разделы стандарта C#, касающиеся наследования.

Лабораторная работа 9. Наследование

В программах требуется описать базовый класс (возможно, абстрактный), в котором с помощью виртуальных или абстрактных методов и свойств задается интерфейс для производных классов. Целью лабораторной работы является максимальное использование наследования, даже если для конкретной задачи оно не дает выигрыша в объеме программы. Во всех классах следует переопределить метод `Equals`, чтобы обеспечить сравнение значений, а не ссылок.

Функция `Main` должна содержать массив из элементов базового класса, заполненный ссылками на производные классы. В этой функции должно демонстрироваться использование всех разработанных элементов классов.

Задание

Создать абстрактный класс Pair (пара значений) с виртуальными арифметическими операциями и методом вывода на экран. На его основе реализовать классы Money (деньги) и Complex (комплексное число).

В классе Money денежная сумма представляется в виде двух целых, в которых хранятся рубли и копейки соответственно. При выводе части числа снабжаются словами "руб." и "коп.". В классе Complex предусмотреть при выводе символ мнимой части (i).

Создать класс Series (набор), содержащий список (или массив) объектов этих классов в динамической памяти. Предусмотреть возможность вывода объектов списка. Написать демонстрационную программу, в которой будут использоваться все методы классов.

Интерфейсы. Контейнерные классы

Описание и использование интерфейсов. Применение стандартных интерфейсов .NET для сравнения, перебора, сортировки и клонирования объектов. Понятие контейнера (коллекции). Использование стандартных коллекций .NET.

Интерфейсы

Презентацию к данной лекции Вы можете скачать ссылка: здесь - http://old.intuit.ru/department/pl/phlcsharp/9/csharp09.ppt.

Синтаксис интерфейса

Интерфейс является "крайним случаем" абстрактного класса. В нем задается набор абстрактных методов, свойств и индексаторов, которые должны быть реализованы в производных классах. Иными словами, интерфейс определяет поведение, которое поддерживается реализующими этот интерфейс классами. Основная идея использования интерфейса состоит в том, чтобы к объектам таких классов можно было обращаться одинаковым образом.

Каждый класс может определять элементы интерфейса по-своему. Так достигается полиморфизм: объекты разных классов по-разному реагируют на вызовы одного и того же метода.

Синтаксис интерфейса аналогичен синтаксису класса:

[атрибуты] [спецификаторы] interface имя_интерфейса [: предки]
тело_интерфейса [;]

Для интерфейса могут быть указаны спецификаторы `new`, `public`, `protected`, `internal` и `private`. Спецификатор `new` применяется для вложенных интерфейсов и имеет такой же смысл, как и соответствующий модификатор метода класса. Остальные спецификаторы управляют видимостью интерфейса. По умолчанию интерфейс доступен только из сборки, в которой он описан (

`internal`).

Интерфейс может наследовать свойства нескольких интерфейсов, в этом случае предки перечисляются через запятую. Тело интерфейса составляют абстрактные методы, шаблоны свойств и индексаторов, а также события.

В качестве примера рассмотрим интерфейс `IAction`, определяющий базовое поведение персонажей компьютерной игры, встречавшихся в предыдущих главах. Допустим, что любой персонаж должен уметь выводить себя на экран, атаковать и красиво умирать:

```
interface IAction
{
    void Draw();
    int Attack(int a);
    void Die();
    int Power { get; }
}
```

В интерфейсе `IAction` заданы заголовки трех методов и шаблон свойства `Power`, доступного только для чтения. Если бы требовалось обеспечить возможность установки свойства, в шаблоне следовало указать ключевое слово `set`, например:

```
int Power { get; set; }
```

Отличия интерфейса от абстрактного класса:

- элементы интерфейса по умолчанию имеют спецификатор доступа `public` и не могут иметь спецификаторов, заданных явным образом;
- интерфейс не может содержать полей и обычных методов — все элементы интерфейса должны быть абстрактными;
- класс, в списке предков которого задается интерфейс, должен определять все его элементы, в то время как потомок абстрактного класса может не переопределять часть абстрактных методов предка (в этом случае производный класс также будет абстрактным);

- класс может иметь в списке предков несколько интерфейсов, при этом он должен определять все их методы.

Реализация интерфейса

В списке предков класса сначала указывается его базовый класс, если он есть, а затем через запятую интерфейсы, которые реализует этот класс. Например, реализация интерфейса `IAction` в классе `Monster` может выглядеть следующим образом:

```
using System;
namespace ConsoleApplication1
{
    interface IAction
    {
        void Draw();
        int Attack( int a );
        void Die();
        int Power { get; }
    }

    class Monster : IAction
    {
        public void Draw()
        {
            Console.WriteLine( "Здесь был " + name );

        }
        public int Attack( int ammo_ )
        {
            ammo -= ammo_;
            if ( ammo > 0 ) Console.WriteLine( "Ба-бах!" );
            else            ammo = 0;
            return ammo;
        }

        public void Die()
        {
```

```
            Console.WriteLine( "Monster " + name + " RIP" );
            health = 0;
        }

        public int Power
        {
            get
            {
                return ammo * health;
            }
        }

        …
    }
```

Сигнатуры методов в интерфейсе и реализации должны полностью совпадать. Для реализуемых элементов интерфейса в классе следует указывать спецификатор `public`. К этим элементам можно обращаться как через объект класса, так и через объект типа соответствующего интерфейса:

```
Monster Vasia = new Monster( 50, 50, "Вася" );  // объект класса Monster
Vasia.Draw();                                   // результат: Здесь был Вася
IAction Actor = new Monster( 10, 10, "Маша" );  // объект типа интерфейса
Actor.Draw();                                   // результат: Здесь был Маша
```

Существует второй способ реализации интерфейса в классе: явное указание имени интерфейса перед реализуемым элементом. Спецификаторы доступа при этом не указываются. К таким элементам можно обращаться в программе только через объект типа интерфейса, например:

```
class Monster : IAction
    {
        int IAction.Power
        {
            get
            {
                return ammo * health;
```

```
        }
    }

    void IAction.Draw()
    {
        Console.WriteLine( "Здесь был " + name );
    }
    ...
}
...
IAction Actor = new Monster( 10, 10, "Маша" );
Actor.Draw();                    // обращение через объект типа интерфейса

// Monster Vasia = new Monster( 50, 50, "Вася" );
// Vasia.Draw();
```

Таким образом, при явном задании имени реализуемого интерфейса соответствующий метод не входит в интерфейс класса. Это позволяет упростить его в том случае, если какие-то элементы интерфейса не требуются конечному пользователю класса.

Работа с объектами через интерфейсы. Операции is и as

При работе с объектом через объект типа интерфейса бывает необходимо убедиться, что объект поддерживает данный интерфейс. Проверка выполняется с помощью бинарной операции `is`. Эта операция определяет, совместим ли текущий тип объекта, находящегося слева от ключевого слова `is`, с типом, заданным справа.

Результат операции равен `true`, если объект можно преобразовать к заданному типу, и `false` в противном случае. Операция обычно используется в следующем контексте:

```
{
    // выполнить преобразование "объекта" к "типу"
    // выполнить действия с преобразованным объектом
}
```

Допустим, мы оформили какие-то действия с объектами в виде метода с параметром типа `object`. Прежде чем использовать этот параметр внутри метода для обращения к методам, описанным в производных классах, требуется выполнить преобразование к производному классу. Для безопасного преобразования следует проверить, возможно ли оно, например так:

```
static void Act( object A )
{
   if ( A is IAction )
   {
      IAction Actor = (IAction) A;
      Actor.Draw();
   }
}
```

В метод `Act` можно передавать любые объекты, но на экран будут выведены только те, которые поддерживают интерфейс `IAction`.

Недостатком использования операции `is` является то, что преобразование фактически выполняется дважды: при проверке и при собственно преобразовании. Более эффективной является другая операция — `as`. Она выполняет преобразование к заданному типу, а если это невозможно, формирует результат `null`, например:

```
static void Act( object A )
{
   IAction Actor = A as IAction;
   if ( Actor != null ) Actor.Draw();
}
```

Обе рассмотренные операции применяются как к интерфейсам, так и к классам.

Интерфейсы и наследование

Интерфейс может не иметь или иметь сколько угодно интерфейсов-предков, в последнем случае он наследует все элементы всех своих

базовых интерфейсов, начиная с самого верхнего уровня.

Как и в обычной иерархии классов, базовые интерфейсы определяют общее поведение, а их потомки конкретизируют и дополняют его. В интерфейсе-потомке можно также указать элементы, переопределяющие унаследованные элементы с такой же сигнатурой. В этом случае перед элементом указывается ключевое слово `new`, как и в аналогичной ситуации в классах. С помощью этого слова соответствующий элемент базового интерфейса скрывается. Класс, реализующий интерфейс, должен определять все его элементы, в том числе унаследованные.

Интерфейс, на собственные или унаследованные элементы которого имеется явная ссылка, должен быть указан в списке предков класса.

Класс наследует все методы своего предка, в том числе те, которые реализовывали интерфейсы. Он может переопределить эти методы с помощью спецификатора `new`, но обращаться к ним можно будет только через объект класса. Если использовать для обращения ссылку на интерфейс, вызывается не переопределенная версия:

```
interface IBase
{
  void A();
}

class Base : IBase
{
  public void A() { ... }
}

class Derived: Base
{
  new public void A() { ... }
}

...
Derived d = new Derived ();
d.A();                    // вызывается Derived.A();
```

```
IBase id = d;
id.A();                    // вызывается Base.A();
```

Однако если интерфейс реализуется с помощью виртуального метода класса, после его переопределения в потомке любой вариант обращения (через класс или через интерфейс) приведет к одному и тому же результату. Метод интерфейса, реализованный явным указанием имени, объявлять виртуальным запрещается.

Существует возможность повторно реализовать интерфейс, указав его имя в списке предков класса наряду с классом-предком, уже реализовавшим этот интерфейс. При этом реализация переопределенных методов базового класса во внимание не принимается:

```
interface IBase
{
  void A();
}

class Base : IBase
{
  void IBase.A() { ... }        // не используется в Derived
}

class Derived : Base, IBase
{
  public void A() { ... }
}
```

Если класс наследует от класса и интерфейса, которые содержат методы с одинаковыми сигнатурами, унаследованный метод класса воспринимается как реализация интерфейса. Вообще при реализации интерфейса учитывается наличие "подходящих" методов в классе независимо от их происхождения. Это могут быть методы, описанные в текущем или базовом классе, реализующие интерфейс явным или неявным образом.

Стандартные интерфейсы .NET

В библиотеке классов .NET определено множество стандартных интерфейсов, задающих желаемое поведение объектов. Например, интерфейс `IComparable` задает метод сравнения объектов на больше-меньше, что позволяет выполнять их сортировку. Реализация интерфейсов `IEnumerable` и `IEnumerator` дает возможность просматривать содержимое объекта с помощью конструкции `foreach`, а реализация интерфейса *`ICloneable`* — клонировать объекты.

Стандартные интерфейсы поддерживаются многими стандартными классами библиотеки. Например, работа с массивами с помощью цикла `foreach` возможна именно потому, что тип `Array` реализует интерфейсы `IEnumerable` и `IEnumerator`. Можно создавать и собственные классы, поддерживающие стандартные интерфейсы, что позволит использовать объекты этих классов стандартными способами.

Сравнение объектов

Интерфейс `IComparable` определен в пространстве имен `System`. Он содержит всего один метод `CompareTo`, возвращающий результат сравнения двух объектов — текущего и переданного ему в качестве параметра:

```
interface IComparable
{
   int CompareTo( object obj )
}
```

Метод должен возвращать:

- 0, если текущий объект и параметр равны;
- отрицательное число, если текущий объект меньше параметра;
- положительное число, если текущий объект больше параметра.

Реализуем интерфейс `IComparable` в знакомом нам классе `Monster`. В качестве критерия сравнения объектов выберем поле `health`. В листинге 9.1 приведена программа, сортирующая массив монстров по возрастанию величины, характеризующей их здоровье.

```
using System;
namespace ConsoleApplication1
{
   class Monster : IComparable
   {
      public Monster( int health, int ammo, string name )
      {
         this.health = health;
         this.ammo   = ammo;
         this.name   = name;
      }

      virtual public void Passport()
      {
         Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",
                        name, health, ammo );
      }

      public int CompareTo( object obj )          // реализация интерфейса
      {
         Monster temp = (Monster) obj;
         if ( this.health > temp.health ) return  1;
         if ( this.health < temp.health ) return -1;
         return 0;
      }

      string name;
      int health, ammo;
   }

   class Class1
   {   static void Main()
      {
         const int n = 3;
         Monster[] stado = new Monster[n];

         stado[0] = new Monster( 50, 50, "Вася" );
         stado[1] = new Monster( 80, 80, "Петя" );
         stado[2] = new Monster( 40, 10, "Маша" );
```

```
            Array.Sort( stado );            // сортировка стала возможной
            foreach ( Monster elem in stado ) elem.Passport();
        }
    }
}
```

Листинг 9.1. Пример реализации интерфейса IComparable

Результат работы программы:

```
Monster Маша    health = 40 ammo = 10
Monster Вася    health = 50 ammo = 50
Monster Петя    health = 80 ammo = 80
```

Во многих алгоритмах требуется выполнять сортировку объектов по различным критериям. В C# для этого используется интерфейс `IComparer`, который мы рассмотрим далее.

Сортировка по разным критериям (интерфейс IComparer)

Интерфейс `IComparer` определен в пространстве имен `System.Collections`. Он содержит один метод `Compare`, возвращающий результат сравнения двух объектов, переданных ему в качестве параметров:

```
interface IComparer
{
    int Compare( object ob1, object ob2 )
}
```

Принцип применения этого интерфейса состоит в там, что для каждого критерия сортировки объектов описывается небольшой вспомогательный класс, реализующий этот интерфейс. Объект этого класса передается в стандартный метод сортировки массива в качестве второго аргумента.

Пример сортировки массива объектов из предыдущего листинга по именам (свойство `Name`, класс `SortByName`) и количеству

вооружений (свойство `Ammo`, класс `SortByAmmo`) приведен в листинге 9.2.

```
using System;
using System.Collections;
namespace ConsoleApplication1
{
   class Monster
   {
      public Monster( int health, int ammo, string name )
      {
         this.health = health;
         this.ammo   = ammo;
         this.name   = name;
      }

      public int Ammo
      {
         get { return ammo; }
         set
         {
            if (value > 0) ammo = value;
            else           ammo = 0;
         }
      }

      public string Name
      {
         get { return name; }
      }

      virtual public void Passport()
      {
         Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",
                        name, health, ammo );
      }

      public class SortByName : IComparer                    //
      {
```

```
        int IComparer.Compare( object ob1, object ob2 )
        {
            Monster m1 = (Monster) ob1;
            Monster m2 = (Monster) ob2;
            return String.Compare( m1.Name, m2.Name );
        }
    }

    public class SortByAmmo : IComparer                    //
    {
        int IComparer.Compare( object ob1, object ob2 )
        {
            Monster m1 = (Monster) ob1;
            Monster m2 = (Monster) ob2;
            if ( m1.Ammo > m2.Ammo ) return  1;
            if ( m1.Ammo < m2.Ammo ) return -1;
            return 0;
        }
    }
    string name;
    int health, ammo;
}

class Class1
{   static void Main()
    {
        const int n = 3;
        Monster[] stado = new Monster[n];

        stado[0] = new Monster( 50, 50, "Вася" );
        stado[1] = new Monster( 80, 80, "Петя" );
        stado[2] = new Monster( 40, 10, "Маша" );

        Console.WriteLine( "Сортировка по имени:" );
        Array.Sort( stado, new Monster.SortByName() );
        foreach ( Monster elem in stado ) elem.Passport();

        Console.WriteLine( "Сортировка по вооружению:" );
        Array.Sort( stado, new Monster.SortByAmmo() );
```

```
            foreach ( Monster elem in stado ) elem.Passport();
        }
    }
}
```

Листинг 9.2. Сортировка по двум критериям

Результат работы программы:

```
Сортировка по имени:
Monster Вася     health = 50 ammo = 50
Monster Маша     health = 40 ammo = 10
Monster Петя     health = 80 ammo = 80
Сортировка по вооружению:
Monster Маша     health = 40 ammo = 10
Monster Вася     health = 50 ammo = 50
Monster Петя     health = 80 ammo = 80
```

Перегрузка операций отношения

Если класс реализует интерфейс `IComparable`, его экземпляры можно сравнивать между собой на больше-меньше. Логично разрешить использовать для этого операции отношения, перегрузив их. Операции должны перегружаться парами: < и >, <= и >=, == и !=. Перегрузка операций обычно выполняется путем делегирования, то есть обращения к переопределенным методам `CompareTo` и `Equals`.

Примечание

Если класс реализует интерфейс `IComparable`, требуется переопределить метод `Equals` и связанный с ним метод `GetHashCode`. Оба метода унаследованы от базового класса `object`.

В листинге 9.3 операции отношения перегружены для класса `Monster`. В качестве критерия сравнения объектов на больше-меньше выступает поле `health`, а при сравнении на равенство попарно сравниваются все поля объектов

```
using System;
```

```
namespace ConsoleApplication1
{
   class Monster : IComparable
   {
      public Monster( int health, int ammo, string name )
      {
         this.health = health;
         this.ammo   = ammo;
         this.name   = name;
      }

      public override bool Equals( object obj )
      {
         if ( obj == null || GetType() != obj.GetType() ) return false;

         Monster temp = (Monster) obj;
         return health == temp.health &&
                ammo   == temp.ammo   &&
                name   == temp.name;
      }

      public override int GetHashCode()
      {
         return name.GetHashCode();
      }

      public static bool operator == ( Monster a, Monster b )
      {
         return a.Equals( b );
      }

      public static bool operator != ( Monster a, Monster b )
      {
         return ! a.Equals( b );
      }

      public static bool operator < ( Monster a, Monster b )
      {
         return ( a.CompareTo( b ) < 0 );
```

```
    }

    public static bool operator > ( Monster a, Monster b )
    {
       return ( a.CompareTo( b ) > 0 );
    }

    public static bool operator <= ( Monster a, Monster b )
    {
       return ( a.CompareTo( b ) <= 0 );
    }

    public static bool operator >= ( Monster a, Monster b )
    {
       return ( a.CompareTo( b ) >= 0 );
    }

    public int CompareTo( object obj )
    {
       Monster temp = (Monster) obj;
       if ( this.health > temp.health ) return  1;
       if ( this.health < temp.health ) return -1;
       return 0;
    }

    string name;
    int health, ammo;
}

class Class1
{   static void Main()
    {
       Monster Вася = new Monster( 70, 80, "Вася" );
       Monster Петя = new Monster( 80, 80, "Петя" );

       if      ( Вася > Петя )  Console.WriteLine( "Вася больше Пети");
       else if ( Вася == Петя ) Console.WriteLine( "Вася == Петя");
       else                     Console.WriteLine( "Вася меньше Пети");
    }
```

```
    }
}
```

Листинг 9.3. Перегрузка операций отношения

Результат работы программы:

```
Вася меньше Пети
```

Клонирование объектов

Клонирование — это создание копии объекта. Копия объекта называется клоном. Как известно, при присваивании одного объекта ссылочного типа другому копируется ссылка, а не сам объект. Если необходимо скопировать в другую область памяти поля объекта, можно воспользоваться методом `MemberwiseClone`, который любой объект наследует от класса `object`. При этом объекты, на которые указывают поля объекта, в свою очередь являющиеся ссылками, не копируются. Это называется поверхностным клонированием.

Для создания полностью независимых объектов необходимо глубокое клонирование, когда в памяти создается дубликат всего дерева объектов, то есть объектов, на которые ссылаются поля объекта, поля полей, и так далее. Алгоритм глубокого клонирования весьма сложен, поскольку требует рекурсивного обхода всех ссылок объекта и отслеживания циклических зависимостей.

Объект, имеющий собственные алгоритмы клонирования, должен объявляться как наследник интерфейса *`ICloneable`* и переопределять его единственный метод `Clone`. В листинге 9.4 приведен пример создания поверхностной копии объекта класса `Monster` с помощью метода `MemberwiseClone`, а также реализован интерфейс *`ICloneable`*. В демонстрационных целях в имя клона объекта добавлено слово "Клон".

```
using System;
namespace ConsoleApplication1
{
    class Monster : ICloneable
```

```
    {
        public Monster( int health, int ammo, string name )
        {
            this.health = health;
            this.ammo   = ammo;
            this.name   = name;
        }
        public Monster ShallowClone()              // поверхностная копия
        {
            return (Monster)this.MemberwiseClone();
        }

        public object Clone()                      // пользовательская копия
        {
            return new Monster(this.health, this.ammo, "Клон " + this.name);
        }

        virtual public void Passport()
        {
            Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",
                               name, health, ammo );
        }

        string name;
        int health, ammo;
    }

    class Class1
    {   static void Main()
        {
            Monster Вася = new Monster( 70, 80, "Вася" );
            Monster X = Вася;
            Monster Y = Вася.ShallowClone();
            Monster Z = (Monster)Вася.Clone();
            ...
        }
    }
}
```

Листинг 9.4. Клонирование объектов

Объект `X` ссылается на ту же область памяти, что и объект `Вася`. Следовательно, если мы внесем изменения в один из этих объектов, это отразится на другом. Объекты `Y` и `Z`, созданные путем клонирования, обладают собственными копиями значений полей и независимы от исходного объекта.

Перебор объектов (интерфейс IEnumerable) и итераторы

Оператор `foreach` является удобным средством перебора элементов объекта. Массивы и все стандартные коллекции библиотеки .NET позволяют выполнять такой перебор благодаря тому, что в них реализованы интерфейсы `IEnumerable` и `IEnumerator`. Для применения оператора `foreach` к пользовательскому типу данных требуется реализовать в нем эти интерфейсы.

Интерфейс `IEnumerable` (перечислимый) определяет всего один метод — `GetEnumerator`, возвращающий объект типа `IEnumerator` (перечислитель), который можно использовать для просмотра элементов объекта.

Интерфейс `IEnumerator` задает три элемента:

- свойство `Current`, возвращающее текущий элемент объекта;
- метод `MoveNext`, продвигающий перечислитель на следующий элемент объекта;
- метод `Reset`, устанавливающий перечислитель в начало просмотра.

Цикл `foreach` использует эти методы для перебора элементов, из которых состоит объект.

Таким образом, если требуется, чтобы для перебора элементов класса мог применяться цикл `foreach`, необходимо реализовать четыре метода: `GetEnumerator`, `Current`, `MoveNext` и `Reset`. Это не интересная работа, а выполнять ее приходится часто, поэтому в версию 2.0 были введены средства, облегчающие выполнение перебора в

объекте — итераторы.

Итератор представляет собой блок кода, задающий последовательность перебора элементов объекта. На каждом проходе цикла `foreach` выполняется один шаг итератора, заканчивающийся выдачей очередного значения. Выдача значения выполняется с помощью ключевого слова *`yield`*.

Рассмотрим создание итератора на примере (листинг 9.5). Пусть требуется создать объект, содержащий боевую группу экземпляров типа `Monster`.

```
using System;
using System.Collections;
namespace ConsoleApplication1
{
   class Monster { ... }
   class Daemon  { ... }
   class Stado : IEnumerable                                  // 1
   {
      private Monster[] mas;
      private int n;

      public Stado()
      {
         mas = new Monster[10];
         n = 0;
      }

      public IEnumerator GetEnumerator()
      {
         for ( int i = 0; i < n; ++i ) yield return mas[i];          // 2
      }

      public void Add( Monster m )
      {
         if ( n >= 10 ) return;
         mas[n] = m;
         ++n;
```

```
        }
    }

    class Class1
    {   static void Main()
        {
            Stado s = new Stado();
            s.Add( new Monster() );
            s.Add( new Monster("Вася") );
            s.Add( new Daemon() );

            foreach ( Monster m in s ) m.Passport();
        }
    }
}
```

Листинг 9.5. Класс с итератором

Все, что требуется сделать в версии 2.0 для поддержки перебора — указать, что класс реализует интерфейс `IEnumerable` (оператор 1), и описать итератор (оператор 2). Доступ к нему может быть осуществлен через методы `MoveNext` и `Current` интерфейса `IEnumerator`.

Преимущество использования итераторов заключается в том, что для одного и того же класса можно задать различный порядок перебора элементов. В листинге 9.6 описаны две дополнительные стратегии перебора элементов класса `Stado`, введенного в листинге 9.5 — перебор в обратном порядке и выборка только тех объектов, которые являются экземплярами класса `Monster`.

```
using System;
using System.Collections;
using MonsterLib;

namespace ConsoleApplication1
{
    class Monster { ... }
    class Daemon  { ... }
    class Stado : IEnumerable
```

```
{
    private Monster[] mas;
    private int n;
    public Stado()
    {
        mas = new Monster[10];
        n = 0;
    }
    public IEnumerator GetEnumerator()
    {
        for ( int i = 0; i < n; ++i ) yield return mas[i];
    }
    public IEnumerable Backwards()              // в обратном порядке
    {
        for ( int i = n - 1; i >= 0; --i ) yield return mas[i];
    }

    public IEnumerable MonstersOnly()           // только монстры
    {
        for ( int i = 0; i < n; ++i )
            if ( mas[i].GetType().Name == "Monster" )
                yield return mas[i];
    }
    public void Add( Monster m )
    {
        if ( n >= 10 ) return;
        mas[n] = m;
        ++n;
    }
}

class Class1
{   static void Main()
    {
        Stado s = new Stado();
        s.Add( new Monster() );
        s.Add( new Monster("Вася") );
        s.Add( new Daemon() );
```

```
            foreach ( Monster i in s )              i.Passport();
            foreach ( Monster i in s.Backwards() )  i.Passport();
            foreach ( Monster i in s.MonstersOnly() ) i.Passport();
        }
    }
}
```

Листинг 9.6. Реализация нескольких стратегий перебора

Теперь, когда вы получили представление об итераторах, рассмотрим их более формально. Блок итератора синтаксически представляет собой обычный блок и может встречаться в теле метода, операции или части `get` свойства, если соответствующее возвращаемое значение имеет тип `IEnumerable` или `IEnumerator`.

В теле блока итератора могут встречаться две конструкции:

- *`yield`* `return` формирует значение, выдаваемое на очередной итерации;
- *`yield`* `break` сигнализирует о завершении итерации.

Ключевое слово *`yield`* имеет специальное значение для компилятора только в этих конструкциях.

Код блока итератора выполняется не так, как обычные блоки. Компилятор формирует служебный объект-перечислитель, при вызове метода `MoveNext` которого выполняется код блока итератора, выдающий очередное значение с помощью ключевого слова *`yield`*. Следующий вызов метода `MoveNext` объекта-перечислителя возобновляет выполнение блока итератора с момента, на котором он был приостановлен в предыдущий раз.

Контейнерные классы

Абстрактные структуры данных

Любая программа предназначена для обработки данных, от способа

организации которых зависит ее алгоритм. Для разных задач необходимы различные способы хранения и обработки данных, поэтому выбор структур данных должен предшествовать созданию алгоритмов и основываться на требованиях к функциональности и быстродействию программы. Наиболее часто в программах используются массив, список, стек, очередь, бинарное дерево, хеш-таблица, граф и множество. Далее дана краткая характеристика каждой из этих структур данных.

Массив — это конечная совокупность однотипных величин. Массив занимает непрерывную область памяти и предоставляет прямой (произвольный) доступ к своим элементам по индексу. Память под массив выделяется до начала работы с ним и впоследствии не изменяется.

В списке каждый элемент связан со следующим и, возможно, с предыдущим. В первом случае список называется односвязным, во втором — двусвязным. Если последний элемент связать указателем с первым, получится кольцевой список. Количество элементов в списке может изменяться в процессе работы программы.

Каждый элемент списка содержит ключ, идентифицирующий этот элемент. Ключ обычно бывает либо целым числом, либо строкой и является частью данных, хранящихся в каждом элементе списка. В качестве ключа в процессе работы со списком могут выступать разные части данных. Например, если создается список из записей, содержащих фамилию, год рождения и стаж работы, любая часть записи может выступать в качестве ключа: при упорядочивании списка по алфавиту ключом будет фамилия, а при поиске, например, ветеранов труда ключом можно сделать стаж. Ключи разных элементов списка могут совпадать.

Над списками можно выполнять операции добавления, удаления и вставки элемента, чтения элемента с заданным ключом, упорядочивания списка по ключу (ключам). Список не обеспечивает произвольный доступ к элементу, поэтому при выполнении операций чтения, вставки и удаления выполняется последовательный перебор элементов, пока не будет найден элемент с заданным ключом.

Стек — частный случай однонаправленного списка, добавление элементов в который и выборка из которого выполняются с одного

конца, называемого вершиной стека. Другие операции со стеком не определены. При выборке элемент исключается из стека. Говорят, что стек реализует принцип обслуживания LIFO (Last In — First Out, последним пришел — первым ушел).

Очередь — частный случай однонаправленного списка, добавление элементов в который выполняется в один конец, а выборка — из другого конца. Другие операции с очередью не определены. При выборке элемент исключается из очереди. Говорят, что очередь реализует принцип обслуживания FIFO (First In — First Out, первым пришел — первым ушел).

Бинарное дерево — динамическая структура данных, состоящая из узлов, каждый из которых содержит, помимо данных, не более двух ссылок на различные бинарные поддеревья. На каждый узел имеется ровно одна ссылка. Начальный узел называется корнем дерева. Пример бинарного дерева приведен на рис. 9.1 (корень обычно изображается сверху). Узел, не имеющий поддеревьев, называется листом. Исходящие узлы называются предками, входящие — потомками. Высота дерева определяется количеством уровней, на которых располагаются его узлы.

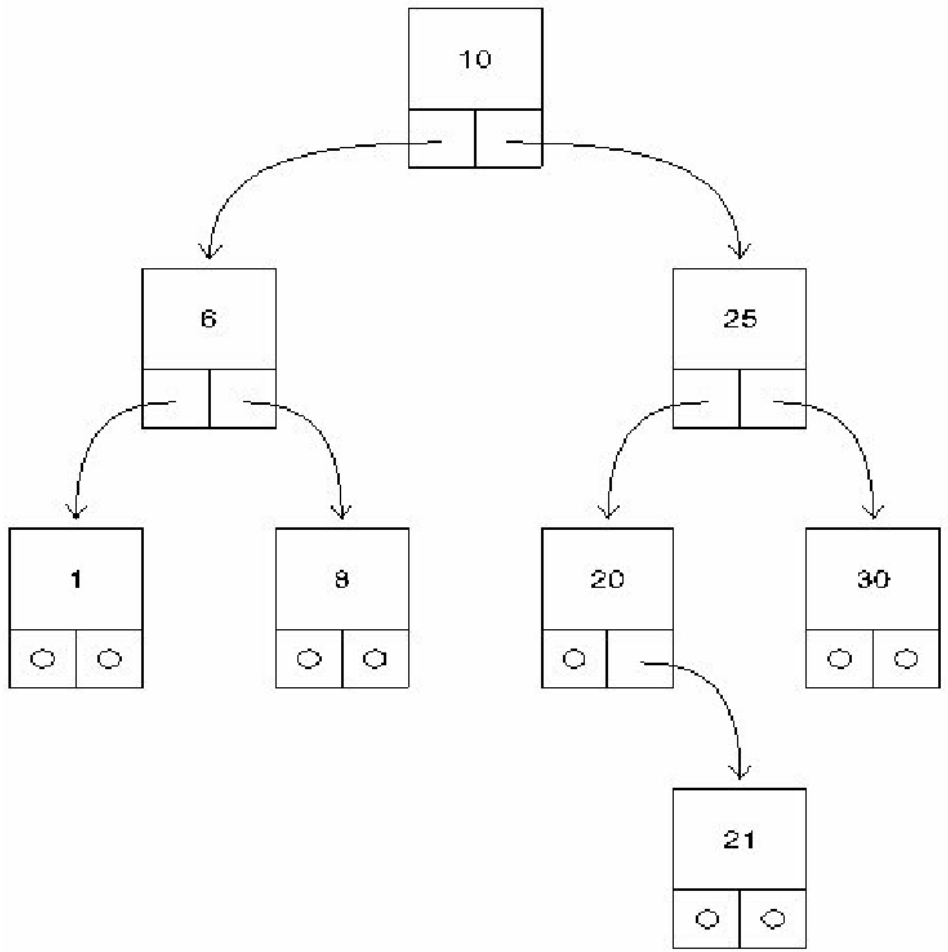


Рис. 9.1. Пример бинарного дерева поиска

Если дерево организовано таким образом, что для каждого узла все ключи его левого поддерева меньше ключа этого узла, а все ключи его правого поддерева — больше, оно называется деревом поиска. Одинаковые ключи не допускаются. В дереве поиска можно найти элемент по ключу, двигаясь от корня и переходя на левое или правое поддерево в зависимости от значения ключа в каждом узле. Такой поиск гораздо эффективнее поиска по списку, поскольку время поиска определяется высотой дерева, а она пропорциональна двоичному логарифму количества узлов.

Хеш-таблица, ассоциативный массив, или словарь — это массив,

доступ к элементам которого осуществляется не по номеру, а по некоторому ключу. Можно сказать, что это таблица, состоящая из пар "ключ-значение" (табл. 9.1). Хеш-таблица эффективно реализует операцию поиска значения по ключу. При этом ключ преобразуется в число (хэш-код), которое используется для быстрого нахождения нужного значения в хеш-таблице.

Таблица 9.1. Пример хэш-таблицы

Ключ	Значение
boy	мальчик
girl	девочка
dog	собачка

Преобразование выполняется с помощью хэш-функции, или функции расстановки. Эта функция обычно производит какие-либо преобразования внутреннего представления ключа. Если хеш-функция распределяет совокупность возможных ключей равномерно по множеству индексов массива, то доступ к элементу по ключу выполняется почти так же быстро, как в массиве.

Смысл хэш-функции состоит в том, чтоб отобразить более широкое множество ключей в более узкое множество индексов. При этом неизбежно возникают так называемые коллизии, когда хеш-функция формирует для двух разных элементов один и тот же хэш-код. В разных реализациях хэш-таблиц используются различные стратегии борьбы с коллизиями.

Граф — это совокупность узлов и ребер, соединяющих различные узлы. Например, можно представить себе карту автомобильных дорог как граф с городами в качестве узлов и шоссе между городами в качестве ребер. Множество реальных практических задач можно описать в терминах графов, что делает их структурой данных, часто используемой при написании программ.

Множество — это неупорядоченная совокупность элементов. Для множеств определены операции проверки принадлежности элемента

множеству, включения и исключения элемента, а также объединения, пересечения и вычитания множеств.

Описанные структуры данных называются абстрактными, поскольку в них не задается реализация допустимых операций.

В библиотеках большинства современных объектно-ориентированных языков программирования представлены стандартные классы, реализующие основные абстрактные структуры данных. Такие классы называются коллекциями, или контейнерами. Для каждого типа коллекции определены методы работы с ее элементами, не зависящие от конкретного типа хранимых данных, поэтому один и тот же вид коллекции можно использовать для хранения данных различных типов. Использование коллекций позволяет сократить сроки разработки программ и повысить их надежность. Изучение возможностей стандартных коллекций и их грамотное применение является необходимым условием создания эффективных и профессиональных программ.

Внимание

Каждый вид коллекции поддерживает свой набор операций над данными, и быстродействие этих операций может быть разным. Выбор вида коллекции зависит от того, что требуется делать с данными в программе и какие требования предъявляются к ее быстродействию. Например, при необходимости часто вставлять и удалять элементы из середины последовательности следует использовать список, а если включение элементов выполняется в конец последовательности — очередь.

В библиотеке .NET определено множество стандартных классов, реализующих большинство перечисленных ранее абстрактных структур данных. Основные пространства имен, в которых описаны эти классы — System.Collections, System.Collections.Specialized и System.Collections.Generic (начиная с версии 2.0).

Пространство имен System.Collections

В пространстве имен System.Collections определены наборы стандартных коллекций и интерфейсов, которые реализованы в этих коллекциях. В таблице 9.2 приведены наиболее важные интерфейсы, часть из которых уже изучались в разделе "Стандартные интерфейсы .NET".

Таблица 9.2. Интерфейсы пространства имен System.Collections

Интерфейс	Назначение
ICollection	Определяет общие характеристики (например, размер) для набора элементов
IComparer	Позволяет сравнивать два объекта
IDictionary	Позволяет представлять содержимое объекта в виде пар "имя-значение"
IDictionaryEnumerator	Используется для нумерации содержимого объекта, поддерживающего интерфейс IDictionary
IEnumerable	Возвращает интерфейс IEnumerator для указанного объекта
IEnumerator	Обычно используется для поддержки оператора foreach в отношении объектов
IHashCodeProvider	Возвращает хэш-код для реализации типа с применением выбранного пользователем алгоритма хэширования
Ilist	Поддерживает методы добавления, удаления и индексирования элементов в списке объектов

В таблице 9.3 перечислены основные коллекции, определенные в пространстве System.Collections.

Таблица 9.3. Коллекции пространства имен System.Collections

Класс	Назначение	Важнейшие из реализованных интерфейсов
ArrayList	Массив, динамически изменяющий свой размер	IList, ICollection, IEnumerable, ICloneable

BitArray	Компактный массив для хранения битовых значений	ICollection, IEnumerable, ICloneable
Hashtable	Хэш-таблица	IDictionary, ICollection, IEnumerable, ICloneable
Queue	Очередь	ICollection, ICloneable, IEnumerable
SortedList	Коллекция, отсортированная по ключам. Доступ к элементам — по ключу или по индексу	IDictionary, ICollection, IEnumerable, ICloneable
Stack	Стек	ICollection, IEnumerable

Пространство имен System.Collections.Specialized включает специализированные коллекции, например, коллекцию строк StringCollection и хэш-таблицу со строковыми ключами StringDictionary.

В качестве примера стандартной коллекции рассмотрим класс ArrayList.

Класс ArrayList

Основным недостатком обычных массивов является то, что объем памяти, необходимый для хранения их элементов, должен быть выделен до начала работы с массивом. Класс ArrayList позволяет программисту не заботиться о выделении памяти и хранить в одном и том же массиве элементы различных типов.

По умолчанию при создании объекта типа ArrayList строится массив из 16 элементов типа object. Можно задать желаемое количество элементов в массиве, передав его в конструктор или установив в качестве значения свойства Capacity, например:

```
ArrayList arr1 = new ArrayList();          // создается массив из 16   элементо
ArrayList arr2 = new ArrayList(1000);  // создается массив из 1000 элемент
ArrayList arr3 = new ArrayList();
arr3.Capacity = 1000;                        // количество элементов задается
```

Класс ArrayList реализован через класс Array, то есть содержит закрытое поле этого класса. Поскольку все типы в C# являются потомками класса object, массив может содержать элементы произвольного типа. Даже если в массиве хранятся обычные целые числа, то есть элементы значимого типа, внутренний класс является массивом ссылок на экземпляры типа object, которые представляют собой упакованный тип-значение. Соответственно, при занесении в массив выполняется упаковка, а при извлечении — распаковка элемента.

Если при добавлении элемента в массив оказывается, что фактическое количество элементов массива превышает его емкость, она автоматически удваивается, то есть происходит повторное выделение памяти и переписывание туда всех существующих элементов. Пример занесения элементов в экземпляр класса ArrayList:

```
arr1.Add( 123 );    arr1.Add( -2 );    arr1.Add( "Вася" );
```

Доступ к элементу выполняется по индексу, однако при этом необходимо явным образом привести полученную ссылку к целевому типу, например:

```
int    a = (int) arr1[0];
int    b = (int) arr1[1];
string s = (string) arr1[2];
```

Попытка приведения к типу, не соответствующему хранимому в элементе, вызывает генерацию исключения InvalidCastException.

Для повышения надежности программ применяется следующий прием: экземпляр класса ArrayList объявляется закрытым полем класса, в котором необходимо хранить коллекцию значений определенного типа, а затем описываются методы работы с этой коллекцией, делегирующие свои функции методам ArrayList.

Недостатком этого решения является то, что для каждого метода

стандартной коллекции приходится описывать метод-оболочку, вызывающий стандартный метод. Хотя это и несложно, но несколько неизящно. В C#, начиная с версии 2.0, появились классы-прототипы (generics), позволяющие решить эту проблему.

Классы-прототипы (generics)

Классы-прототипы (generics) — это классы, имеющие в качестве параметров типы данных. Чаще всего их применяют для хранения данных, то есть в качестве контейнерных классов, или коллекций. Во вторую версию библиотеки .NET добавлены параметризованные коллекции для представления основных структур данных, применяющихся при создании программ — стека, очереди, списка, словаря и т. д. Эти коллекции, расположенные в пространстве имен System.Collections.Generic, дублируют аналогичные коллекции пространства имен System.Collections. В таблице 9.4 приводится соответствие между обычными и параметризованными коллекциями библиотеки .NET (параметры, определяющие типы данных, хранимых в коллекции, указаны в угловых скобках).

Таблица 9.4. Параметризованные коллекции библиотеки .NET версии 2.0

Класс-прототип (версия 2.0)	Обычный класс
Comparer<T>	Comparer
Dictionary<K,T>	HashTable
LinkedList<T>	—
List<T>	ArrayList
Queue<T>	Queue
SortedDictionary<K,T>	SortedList
Stack<T>	Stack<T>

В качестве примера рассмотрим применение универсального "двойника" класса ArrayList — класса List<T> — для хранения коллекции объектов известных нам классов Monster и Daemon, а также для хранения целых чисел.

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

namespace ConsoleApplication1
{
   using MonsterLib;    // библиотека, в которой хранятся классы Monster и
   class Program
   {
      static void Main()
      {
         List<Monster> stado = new List<Monster>();
         stado.Add( new Monster( "Monia" ) );
         stado.Add( new Monster( "Monk" ) );
         stado.Add( new Daemon ( "Dimon", 3 ) );

         foreach ( Monster x in stado ) x.Passport();

         List<int> lint = new List<int>();
         lint.Add( 5 ); lint.Add( 1 ); lint.Add( 3 );
         lint.Sort();
         int a = lint[2];
         Console.WriteLine( a );

         foreach ( int x in lint ) Console.Write( x + " ");
}}}
```

Листинг 9.7. Использование универсальной коллекции List<T>

Результат работы программы:

```
Monster Monia    health = 100 ammo = 100
Monster Monk     health = 100 ammo = 100
Daemon Dimon     health = 100 ammo = 100 brain = 3
5
1 3 5
```

В листинге 9.7 две коллекции. Первая (stado) содержит элементы пользовательских классов, которые находятся в библиотеке MonsterLib.dll. В коллекции, для которой объявлен тип элементов

Monster, благодаря полиморфизму можно хранить элементы любого производного класса, но не элементы других типов. Достоинством такого ограничения является то, что компилятор может выполнить контроль типов, что повышает надежность программы и упрощает поиск ошибок.

Коллекция lint состоит из целых чисел, причем для работы с ними не требуются ни операции упаковки и распаковки, ни явные преобразования типа при получении элемента из коллекции.

Классы-прототипы называют также родовыми или шаблонными, поскольку они представляют собой образцы, по которым во время выполнения программы строятся конкретные классы.

Вопросы и задания для самостоятельной работы студента

1. Изучите разделы стандарта C#, касающиеся интерфейсов.
2. Изучите по справочной системе основные стандартные интерфейсы .NET.
3. Чем интерфейс отличается от абстрактного класса?
4. Должен ли класс реализовывать все методы всех своих интерфейсов-предков?
5. Какие операции используются для проверки, реализует ли класс заданный интерфейс?
6. Перечислите стандартные интерфейсы .NET, которые определяют возможности сортировки, клонирования и просмотра объектов с помощью оператора foreach.
7. Опишите, как используется конструкция yield.
8. Изучите по справочной системе состав пространства имен System.Collections.
9. Изучите по справочной системе свойства и методы класса ArrayList.
10. Изучите разделы стандарта C#, касающиеся классов-прототипов.
11. Изучите по справочной системе состав пространства имен System.Collections.Generic. Опишите основные виды абстрактных структур данных.
12. Допускает ли структура "линейный список" прямой доступ к

своим элементам?
13. В чем основная задача хеш-функции?
14. В каких задачах используется стек? Приведите примеры.
15. Сравните время поиска элемента в линейном списке и в дереве поиска.
16. В чем преимущество массива перед другими структурами данных?
17. Опишите состав пространства имен System.Collections и дайте характеристику основных типов-коллекций.
18. Какие новые контейнеры появились в библиотеке классов для версии C# 2.0?
19. В чем преимущества контейнеров версии C# 2.0?
20. Какие методы содержит класс ArrayList?

Лабораторная работа 10. Интерфейсы и параметризованные коллекции

Выполнить задания лабораторной работы "Наследование классов", используя для хранения экземпляров разработанных классов стандартные параметризованные коллекции. Во всех классах реализовать интерфейс `IComparable` и перегрузить операции отношения для реализации значимой семантики сравнения объектов по какому-либо полю на усмотрение студента.

Делегаты и события

Назначение, описание и использование делегатов. Паттерн "наблюдатель". Механизм событий. Введение в многопоточные приложения. Асинхронные делегаты.

Делегаты

Презентацию к данной лекции Вы можете скачать ссылка: здесь - http://old.intuit.ru/department/pl/phlcsharp/10/csharp10.ppt.

Делегат — это вид класса, предназначенный для хранения ссылок на методы. Делегат, как любой класс, можно передать в качестве параметра, а затем вызвать инкапсулированный в нем метод. Делегаты используются для поддержки событий, а также как самостоятельная конструкция языка (в том числе в многопоточных приложениях).

Описание делегатов

Описание делегата задает сигнатуру методов, которые могут быть вызваны с его помощью:

```
[ атрибуты ] [ спецификаторы ] delegate тип имя_делегата ( [ параметры
```

Допускаются спецификаторы `new`, `public`, `protected`, `internal` и `private`. Тип описывает возвращаемое значение методов, вызываемых с помощью делегата, а необязательными параметрами делегата являются параметры этих методов. Делегат может хранить ссылки на несколько методов и вызывать их поочередно; естественно, что сигнатуры всех методов должны совпадать.

Пример описания делегата:

```
public delegate void D ( int i );
```

Здесь описан тип делегата, который может хранить ссылки на методы, возвращающие `void` и принимающие один параметр целого типа. Объявление делегата можно размещать непосредственно в

пространстве имен или внутри класса.

Использование делегатов

Чтобы воспользоваться делегатом, необходимо создать его экземпляр и задать имена методов, на которые он будет ссылаться. При вызове экземпляра делегата вызываются все заданные в нем методы.

Делегаты применяются в основном для следующих целей:

- получения возможности определять вызываемый метод не при компиляции, а во время выполнения программы;
- обеспечения связи между объектами по типу "источник — наблюдатель";
- создания универсальных методов, в которые можно передавать другие методы;
- поддержки механизма обратных вызовов.

Рассмотрим сначала пример реализации первой из этих целей. В листинге 10.1 объявляется делегат, с помощью которого один и тот же оператор используется для вызова двух разных методов (`C00l` и *`Hack`*).

```
using System;
namespace ConsoleApplication1
{
   delegate void Del ( ref string s );              // объявление делегата

   class Class1
   {
      public static void C00l ( ref string s )              // метод 1
      {
         string temp = "";
         for ( int i = 0; i < s.Length; ++i )
         {
            if      ( s[i] == 'o' || s[i] == 'O') temp += '0';
            else if ( s[i] == 'l' )               temp += '1';
            else                                  temp += s[i];
```

```
            }
            s = temp;
        }

        public static void Hack ( ref string s )                // метод 2
        {
            string temp = "";
            for ( int i = 0; i < s.Length; ++i )
                if ( i / 2 * 2 == i ) temp += char.ToUpper( s[i] );
                else                  temp += s[i];

            s = temp;
        }

        static void Main()
        {
            string s = "cool hackers";
            Del d;                                  // экземпляр делегата

            for ( int i = 0; i < 2; ++i )
            {
                d = new Del( C00l );                // инициализация методом 1
                if ( i == 1 ) d = new Del(Hack);  // инициализация методом 2

                d( ref s );     // использование делегата для вызова методов
                Console.WriteLine( s );
            }
        }
    }
}
```

Листинг 10.1. Простейшее использование делегата

Результат работы программы:

```
c001 hackers
C001 hAcKeRs
```

Использование делегата имеет тот же синтаксис, что и вызов метода. Если делегат хранит ссылки на несколько методов, они вызываются

последовательно в том порядке, в котором были добавлены в делегат.

Добавление метода в список выполняется либо с помощью метода `Combine`, унаследованного от класса `System.Delegate`, либо, что удобнее, с помощью перегруженной операции сложения. Вот как выглядит измененный метод `Main` из предыдущего листинга, в котором одним вызовом делегата выполняется преобразование исходной строки сразу двумя методами.

```
static void Main()
{
        string s = "cool hackers";
        Del d = new Del( C00l );
        d += new Del( Hack );              // добавление метода в делегат

        d( ref s );
        Console.WriteLine( s );           // результат: C001 hAcKeRs
}
```

При вызове последовательности методов с помощью делегата необходимо учитывать следующее:

- сигнатура методов должна в точности соответствовать делегату;
- методы могут быть как статическими, так и обычными методами класса;
- каждому методу в списке передается один и тот же набор параметров;
- если параметр передается по ссылке, изменения параметра в одном методе отразятся на его значении при вызове следующего метода;
- если параметр передается с ключевым словом `out` или метод возвращает значение, результатом выполнения делегата является значение, сформированное последним из методов списка (в связи с этим рекомендуется формировать списки только из делегатов, имеющих возвращаемое значение типа `void`);
- если в процессе работы метода возникло исключение, не обработанное в том же методе, последующие методы в списке не выполняются, а происходит поиск обработчиков в объемлющих делегат блоках;

- попытка вызвать делегат, в списке которого нет ни одного метода, вызывает генерацию исключения `System.NullReferenceException`.

Паттерн "наблюдатель"

Рассмотрим применение делегатов для обеспечения связи между объектами по типу "источник — наблюдатель". В результате разбиения системы на множество совместно работающих классов появляется необходимость поддерживать согласованное состояние взаимосвязанных объектов. При этом желательно избежать жесткой связанности классов, так как это часто негативно сказывается на возможности многократного использования кода.

Для обеспечения связи между объектами во время выполнения программы применяется следующая стратегия. Объект, называемый источником, при изменении своего состояния, которое может представлять интерес для других объектов, посылает им уведомления. Эти объекты называются наблюдателями. Получив уведомление, наблюдатель опрашивает источник, чтобы синхронизировать с ним свое состояние. Примером такой стратегии может служить связь электронной таблицы с созданными на ее основе диаграммами.

Программисты часто используют одну и ту же схему организации и взаимодействия объектов в разных контекстах. За такими схемами закрепилось название паттерны, или шаблоны проектирования. Описанная стратегия известна под названием паттерн "наблюдатель".

Наблюдатель (observer) определяет между объектами зависимость типа "один ко многим", так что при изменении состояния одного объекта все зависящие от него объекты получают извещение и автоматически обновляются. Рассмотрим пример (листинг 10.2), в котором демонстрируется схема оповещения источником трех наблюдателей. Гипотетическое изменение состояния объекта моделируется сообщением "ОЙ!". Один из методов в демонстрационных целях сделан статическим.

```
using System;
```

```
namespace ConsoleApplication1
{
   public delegate void Del( object o );            // объявление делегата

   class Subj                                       // класс-источник
   {
      Del dels;                        // объявление экземпляра делегата

      public void Register( Del d )                 // регистрация делегата
      {
         dels += d;
      }

      public void OOPS()                            // что-то произошло
      {
         Console.WriteLine( "ОЙ!" );
         if ( dels != null ) dels( this );     // оповещение наблюдателей
      }
   }

   class ObsA                                       // класс-наблюдатель
   {
      public void Do( object o )           // реакция на событие источника
      {
         Console.WriteLine( "Бедняжка!" );
      }
   }

   class ObsB                                       // класс-наблюдатель
   {
      public static void See( object o )   // реакция на событие источника
      {
         Console.WriteLine( "Да ну, ерунда!" );
      }
   }

   class Class1
   {
      static void Main()
```

```
        {
            Subj s  = new Subj();              //  объект класса-источника

            ObsA o1 = new ObsA();              //              объекты
            ObsA o2 = new ObsA();              //      класса-наблюдателя

            s.Register( new Del( o1.Do ) );    //      регистрация методов
            s.Register( new Del( o2.Do ) );    // наблюдателей в источнике
            s.Register( new Del( ObsB.See ) );  //  ( экземпляры делегата )

            s.OOPS();                          //    инициирование события
        }
    }
}
```

Листинг 10.2. Оповещение наблюдателей с помощью делегата

В источнике объявляется экземпляр делегата, в этот экземпляр заносятся методы тех объектов, которые хотят получать уведомление об изменении состояния источника. Этот процесс называется регистрацией делегатов. При регистрации имя метода добавляется к списку. При наступлении "часа X" все зарегистрированные методы поочередно вызываются через делегат.

Результат работы программы:

```
ОЙ!
Бедняжка!
Бедняжка!
Да ну, ерунда!
```

Для обеспечения обратной связи между наблюдателем и источником делегат объявлен с параметром типа `object`, через который в вызываемый метод передается ссылка на вызывающий объект. Следовательно, в вызываемом методе можно получать информацию о состоянии вызывающего объекта и посылать ему сообщения.

Связь "источник — наблюдатель" устанавливается во время выполнения программы для каждого объекта по отдельности. Если наблюдатель больше не хочет получать уведомления от источника,

можно удалить соответствующий метод из списка делегата с помощью метода `Remove` или перегруженной операции вычитания, например:

```
public void UnRegister( Del d )              // удаление делегата
{
   dels -= d;
}
```

Операции

Делегаты можно сравнивать на равенство и неравенство. Два делегата равны, если они оба не содержат ссылок на методы или если они содержат ссылки на одни и те же методы в одном и том же порядке. Сравнивать можно даже делегаты различных типов при условии, что они имеют один и тот же тип возвращаемого значения и одинаковые списки параметров.

С делегатами одного типа можно выполнять операции простого и сложного присваивания, например:

```
Del d1 = new Del( o1.Do );      // o1.Do
Del d2 = new Del( o2.Do );      // o2.Do
Del d3 = d1 + d2;               // o1.Do и o2.Do
d3 += d1;                       // o1.Do, o2.Do и o1.Do
d3 -= d2;                       // o1.Do и o1.Do
```

Эти операции могут понадобиться, например, в том случае, если в разных обстоятельствах требуется вызывать разные наборы и комбинации наборов методов.

Делегат, как и строка `string`, является неизменяемым типом данных, поэтому при любом изменении создается новый экземпляр, а старый впоследствии удаляется сборщиком мусора.

Передача делегатов в методы

Поскольку делегат является классом, его можно передавать в методы в

качестве параметра. Таким образом обеспечивается функциональная параметризация: в метод можно передавать не только различные данные, но и различные функции их обработки. Функциональная параметризация применяется для создания универсальных методов и обеспечения возможности обратного вызова.

В качестве простейшего примера универсального метода можно привести метод вывода таблицы значений функции, в который передается диапазон значений аргумента, шаг его изменения и вид вычисляемой функции. Пример приведен в листинге 10.3.

```
using System;
namespace ConsoleApplication1
{
   public delegate double Fun( double x );          // объявление делегата

   class Class1
   {
      public static void Table( Fun F, double x, double b )
      {
         Console.WriteLine( " ----- X ----- Y -----" );
         while (x <= b)
         {
            Console.WriteLine( "| {0,8:0.000} | {1,8:0.000} |", x, F(x));
            x += 1;
         }
         Console.WriteLine( " ----------------------" );
      }

      public static double Simple( double x )
      {
         return 1;
      }

      static void Main()
      {
         Console.WriteLine( " Таблица функции Sin " );
         Table( new Fun( Math.Sin ), -2, 2 );
```

```
            Console.WriteLine( " Таблица функции Simple " );
            Table( new Fun( Simple ), 0, 3 );
        }
    }
}
```

Листинг 10.3. Передача делегата через список параметров

Результат работы программы:

```
 Таблица функции Sin
 ----- X ----- Y -----
|  -2,000 |  -0,909 |
|  -1,000 |  -0,841 |
|   0,000 |   0,000 |
|   1,000 |   0,841 |
|   2,000 |   0,909 |
 --------------------
  Таблица функции Simple
 ----- X ----- Y -----
|   0,000 |   1,000 |
|   1,000 |   1,000 |
|   2,000 |   1,000 |
|   3,000 |   1,000 |
 --------------------
```

Обратный вызов (callback) представляет собой вызов функции, передаваемой в другую функцию в качестве параметра. Рассмотрим рисунок 10.1. Допустим, в библиотеке описана функция `A`, параметром которой является имя другой функции. В вызывающем коде описывается функция с требуемой сигнатурой (`B`) и передается в функцию `A`. Выполнение функции `A` приводит к вызову `B`, то есть управление передается из библиотечной функции обратно в вызывающий код.

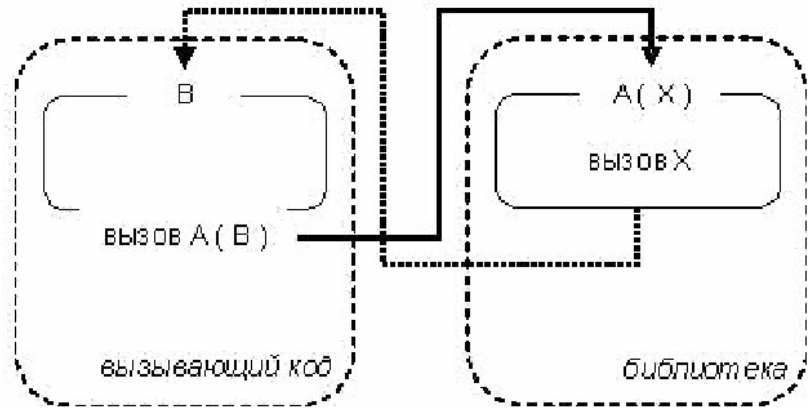


Рис. 10.1. Механизм обратного вызова

Механизм обратного вызова широко используется в программировании. Например, он реализуется во многих стандартных функциях Windows.

Начиная с Visual Studio 2005, использующей версию 2.0 языка C#, можно применять упрощенный синтаксис для делегатов. Первое упрощение заключается в том, что в большинстве случаев явным образом создавать экземпляр делегата не требуется, поскольку он создается автоматически по контексту. Второе упрощение заключается в возможности создания так называемых анонимных методов — фрагментов кода, описываемых непосредственно в том месте, где используется делегат:

```
static void Main()
        {
            Console.WriteLine( " Таблица функции Sin " );
            Table( Math.Sin, -2, 2 );                    // упрощение 1

            Console.WriteLine( " Таблица функции Simple " );
            Table( delegate (double x ){ return 1; }, 0, 3 ); // упрощение 2
        }
    }
}
```

В первом случае экземпляр делегата, соответствующего функции `Sin`, создается автоматически. Чтобы это могло произойти, список параметров и тип возвращаемого значения функции должны быть совместимы с делегатом. Во втором случае не требуется оформлять

простой фрагмент кода в виде отдельной функции `Simple`, как это было сделано в предыдущем листинге, — код функции оформляется как анонимный метод и встраивается прямо в место передачи.

События

Событие — это элемент класса, позволяющий ему посылать другим объектам уведомления об изменении своего состояния. При этом для объектов, являющихся наблюдателями события, активизируются методы-обработчики этого события. Обработчики должны быть зарегистрированы в объекте-источнике события. Таким образом, механизм событий формализует на языковом уровне паттерн "наблюдатель", который рассматривался в предыдущем разделе.

Механизм событий можно также описать с помощью модели "публикация — подписка": один класс, являющийся отправителем (sender) сообщения, публикует события, которые он может инициировать, а другие классы, являющиеся получателями (receivers) сообщения, подписываются на получение этих событий.

События построены на основе делегатов: с помощью делегатов вызываются методы-обработчики событий. Поэтому создание события в классе состоит из следующих частей:

- описание делегата, задающего сигнатуру обработчиков событий;
- описание события;
- описание метода (методов), инициирующих событие.

Синтаксис события похож на синтаксис делегата:

```
[ атрибуты ] [ спецификаторы ] event тип имя_события
```

Для событий применяются спецификаторы `new`, `public`, `protected`, `internal`, `private`, `static`, `virtual`, `sealed`, `override`, `abstract` и `extern`. Например, так же как и методы, событие может быть статическим (`static`), тогда оно связано с классом в целом, или обычным — в этом случае оно связано с экземпляром класса. Тип события — это тип делегата, на котором

основано событие.

Пример описания делегата и соответствующего ему события:

```
public delegate void Del( object o );              // объявление делегата
class A
{
   public event Del Oops;                          // объявление события
   ...
}
```

Обработка событий выполняется в классах-получателях сообщения. Для этого в них описываются методы-обработчики событий, сигнатура которых соответствует типу делегата. Каждый объект (не класс!), желающий получать сообщение, должен зарегистрировать в объекте-отправителе этот метод.

Как видите, это в точности тот же самый механизм, который рассматривался в предыдущем разделе. Единственное отличие состоит в том, что при использовании событий не требуется описывать метод, регистрирующий обработчики, поскольку события поддерживают операции += и −=, добавляющие обработчик в список и удаляющие его из списка.

В листинге 10.4 приведен код из листинга 10.2, переработанный с использованием событий, а рисунок 10.2 поясняет организацию обработки событий.

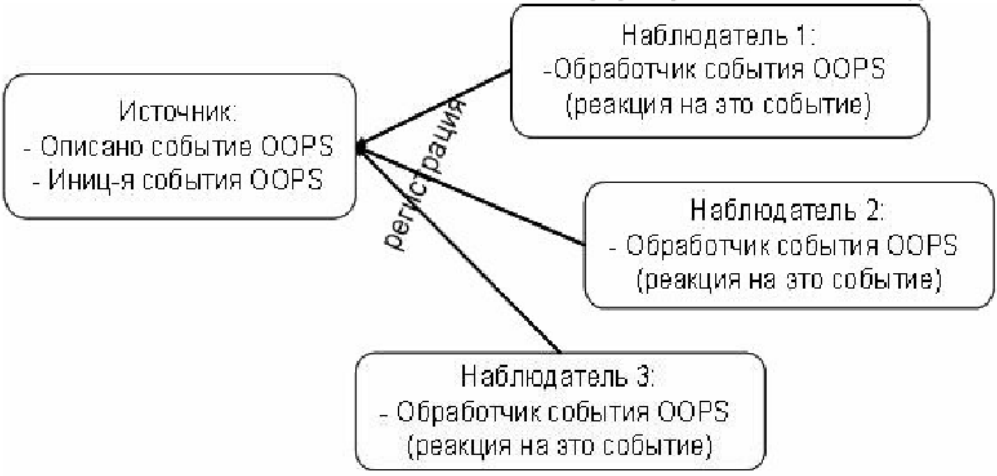


Рис. 10.2. Выполнение программы с двумя нулевыми коэффициентами

```
using System;
namespace ConsoleApplication1
{
   public delegate void Del();                    // объявление делегата

   class Subj                                     // класс-источник
   {
      public event Del Oops;                      // объявление события

      public void CryOops()                       // метод, инициирующий событие
      {
         Console.WriteLine( "ОЙ!" );
         if ( Oops != null ) Oops();
      }
   }

   class ObsA                                     // класс-наблюдатель
   {
      public void Do();                           // реакция на событие источника
      {
         Console.WriteLine( "Бедняжка!" );
      }
   }
```

```
    class ObsB                              // класс-наблюдатель
    {
       public static void See()             // реакция на событие источника
       {
          Console.WriteLine( "Да ну, ерунда!" );
       }
    }

    class Class1
    {
       static void Main()
       {
          Subj s  = new Subj();             //   объект класса-источника

          ObsA o1 = new ObsA();             //                объекты
          ObsA o2 = new ObsA();             //       класса-наблюдателя

          s.Oops += new Del( o1.Do );       //              добавление
          s.Oops += new Del( o2.Do );       //             обработчиков
          s.Oops += new Del( ObsB.See );    //                к событию

          s.CryOops();                      //     инициирование события
       }
    }
}
```

Листинг 10.4. Оповещение наблюдателей с помощью событий

Внешний код может работать с событиями единственным образом: добавлять обработчики в список или удалять их, поскольку вне класса могут использоваться только операции `+=` и `-=`. Тип результата этих операций — `void`, в отличие от операций сложного присваивания для арифметических типов.

Внутри класса, в котором описано событие, с ним можно обращаться, как с обычным полем, имеющим тип делегата: использовать операции отношения, присваивания и т. д. Значение события по умолчанию — `null`.

В библиотеке .NET описано огромное количество стандартных делегатов, предназначенных для реализации механизма обработки событий. Большинство этих классов оформлено по одним и тем же правилам:

- имя делегата заканчивается суффиксом `EventHandler`;
- делегат получает два параметра:
  - первый параметр задает источник события и имеет тип `object`;
  - второй параметр задает аргументы события и имеет тип `EventArgs` или производный от него.

Если обработчикам события требуется специфическая информация о событии, то для этого создают класс, производный от стандартного класса `EventArgs`, и добавляют в него необходимую информацию. Если делегат не использует такую информацию, можно обойтись стандартным классом делегата `System.EventHandler`.

Имя обработчика события принято составлять из префикса `On` и имени события. В листинге 10.5 приведен пример из листинга 10.4, оформленный в соответствии со стандартными соглашениями .NET.

```
using System;
namespace ConsoleApplication1
{
   class Subj
   {
      public event EventHandler Oops;

      public void CryOops()
      {
         Console.WriteLine( "ОЙ!" );
         if ( Oops != null ) Oops( this, null );
      }
   }

   class ObsA
   {
      public void OnOops( object sender, EventArgs e )
```

```
        {
            Console.WriteLine( "Бедняжка!" );
        }
    }

    class ObsB
    {
        public static void OnOops( object sender, EventArgs e )
        {
            Console.WriteLine( "Да ну, ерунда!" );
        }
    }

    class Class1
    {   static void Main()
        {
            Subj s  = new Subj();

            ObsA o1 = new ObsA();
            ObsA o2 = new ObsA();

            s.Oops += new EventHandler( o1.OnOops );
            s.Oops += new EventHandler( o2.OnOops );
            s.Oops += new EventHandler( ObsB.OnOops );

            s.CryOops();
        }
    }
}
```

Листинг 10.5. Использование стандартного делегата EventHandler

Те, кто работает с C# версии начиная с 2.0, могут упростить эту программу, используя возможность неявного создания делегатов при регистрации обработчиков событий. Об этом можно прочитать в учебнике [4].

Многопоточные приложения

Приложение .NET состоит из одного или нескольких процессов. Процессу принадлежат выделенная для него область оперативной памяти и ресурсы. Каждый процесс может состоять из нескольких доменов (частей) приложения, ресурсы которых изолированы друг от друга. В рамках домена может быть запущено несколько потоков выполнения. Поток (thread) представляет собой часть исполняемого кода программы. Иногда термин "thread" переводится буквально — "нить", чтобы отличить его от потоков ввода-вывода. Поэтому в литературе можно встретить и термин "многонитевые приложения".

В каждом процессе есть первичный поток, исполняющий роль точки входа в приложение. Для консольных приложений это метод `Main`.

Многопоточные приложения создают как для многопроцессорных, так и для однопроцессорных систем. Основной целью при этом являются повышение общей производительности и сокращение времени реакции приложения. Управление потоками осуществляет операционная система. Каждый поток получает некоторое количество квантов времени, по истечении которого управление передается другому потоку. Это создает у пользователя однопроцессорной машины впечатление одновременной работы нескольких потоков и позволяет, к примеру, выполнять ввод текста одновременно с длительной операцией по передаче данных.

Недостатки многопоточности:

- большое количество потоков ведет к увеличению накладных расходов, связанных с переключением потоков, что снижает общую производительность;
- возникают проблемы синхронизации данных, связанные с потенциальной возможностью доступа к одним и тем же данным со стороны нескольких потоков (например, если один поток начинает изменение общих данных, а отведенное ему время истекает, доступ к этим же данным может получить другой поток, который, изменяя данные, необратимо их повреждает).

Класс Thread

В .NET многопоточность поддерживается в основном с помощью пространства имен `System.Threading`. Некоторые типы этого пространства описаны в таблице 10.1.

Таблица 10.1. Некоторые типы пространства имен System.Threading

Тип	Описание
`Monitor`	Класс, обеспечивающий синхронизацию доступа к объектам
`Mutex`	Класс-примитив синхронизации, который используется также для синхронизации между процессами
`Thread`	Класс, который создает поток, устанавливает его приоритет, получает информацию о состоянии
`ThreadPool`	Класс, используемый для управления набором взаимосвязанных потоков — пулом потоков
`Timer`	Класс, определяющий механизм вызова заданного метода в заданные интервалы времени для пула потоков
`WaitHandle`	Класс, инкапсулирующий объекты синхронизации, которые ожидают доступа к разделяемым ресурсам
`ThreadStart`	Делегат, представляющий метод, который должен быть выполнен при запуске потока
`TimerCallback`	Делегат, представляющий метод, обрабатывающий вызовы от класса `Timer`
`WaitCallback`	Делегат, представляющий метод для элементов класса `ThreadPool`
`ThreadPriority`	Перечисление, описывающее приоритет потока
`ThreadState`	Перечисление, описывающее состояние потока

Первичный поток создается автоматически. Для запуска вторичных потоков используется класс `Thread`. При создании объекта-потока ему передается делегат, определяющий метод, выполнение которого выделяется в отдельный поток:

```
Thread t = new Thread ( new ThreadStart( имя_метода ) );
```

После создания потока заданный метод начинает в нем свою работу, а первичный поток продолжает выполняться. В листинге 10.6 приведен пример одновременной работы двух потоков.

```
using System;
using System.Threading;
namespace ConsoleApplication1
{    class Program
   {
      static public void Hedgehog()         // метод для вторичного потока
      {
         for ( int i = 0; i < 6; ++i )
         {
            Console.Write(" " + i ); Thread.Sleep( 1000 );
         }
      }

      static void Main()
      {
         Console.WriteLine( "Первичный поток " +
            Thread.CurrentThread.GetHashCode() );

         Thread ta = new Thread( new ThreadStart(Hedgehog) );
         Console.WriteLine( "Вторичный поток " + ta.GetHashCode() );
         ta.Start();

         for ( int i = 0; i > -6; --i )
         {
            Console.Write( " " + i ); Thread.Sleep( 400 );
         }
      }
   }
}
```

Листинг 10.6. Создание вторичного потока

Результат работы программы:

```
Первичный поток 1
Вторичный поток 2
 0 0 -1 -2 1 -3 -4 2 -5 3 4 5
```

В листинге используется метод `Sleep`, останавливающий функционирование потока на заданное количество миллисекунд. Как видите, оба потока работают одновременно. Если бы они работали с одним и тем же файлом, он был бы испорчен так же, как и приведенный вывод на консоль, поэтому такой способ распараллеливания вычислений имеет смысл только для работы с различными ресурсами.

В таблице 10.2 перечислены основные элементы класса `Thread`.

Таблица 10.2. Основные элементы класса Thread

Элемент	Вид	Описание
`CurrentThread`	Статическое свойство	Возвращает ссылку на выполняющийся поток (только для чтения)
`Name`	Свойство	Установка текстового имени потока
`Priority`	Свойство	Получить/установить приоритет потока (используются значения перечисления `ThreadPriority`)
`ThreadState`	Свойство	Возвращает состояние потока (используются значения перечисления `ThreadState`)
`Abort`	Метод	Генерирует исключение `ThreadAbortException`. Вызов этого метода обычно завершает работу потока
`Sleep`	Статический метод	Приостанавливает выполнение текущего потока на заданное количество миллисекунд
`Interrupt`	Метод	Прерывает работу текущего потока
`Join`	Метод	Блокирует вызывающий поток до завершения другого потока или

`Join`	Метод	указанного промежутка времени и завершает поток
`Resume`	Метод	Возобновляет работу после приостановки потока
`Start`	Метод	Начинает выполнение потока, определенного делегатом `ThreadStart`
`Suspend`	Метод	Приостанавливает выполнение потока. Если выполнение потока уже приостановлено, то игнорируется

Можно создать несколько потоков, которые будут совместно использовать один и тот же код. Пример приведен в листинге 10.7.

```
using System;
using System.Threading;
namespace ConsoleApplication1
{
   class Class1
   {   public void Do()
      {
         for ( int i = 0; i < 4; ++i )
            { Console.Write( " " + i ); Thread.Sleep( 3 ); }
      }
   }

   class Program
   {   static void Main()
      {
         Class1 a = new Class1();
         Thread t1 = new Thread( new ThreadStart( a.Do ) );
         t1.Name = "Second";
         Console.WriteLine( "Поток " + t1.Name );
         t1.Start();

         Thread t2 = new Thread( new ThreadStart( a.Do ) );
         t2.Name = "Third";
```

```
            t2.Start();
        }
    }
}
```

Листинг 10.7. Потоки, использующие один объект

Результат работы программы:

```
Поток Second
Поток Third
 0 0 1 1 2 2 3 3
```

Варианты вывода могут несколько различаться, поскольку один поток прерывает выполнение другого в неизвестные моменты времени.

Для того чтобы блок кода мог использоваться в каждый момент только одним потоком, применяется оператор `lock`. Формат оператора:

```
lock ( выражение ) блок_операторов
```

Выражение определяет объект, который требуется заблокировать. Для обычных методов в качестве выражения используется ключевое слово `this`, для статических — `typeof( класс )`. Блок операторов задает критическую секцию кода, которую требуется заблокировать.

Например, блокировка операторов в приведенном ранее методе `Do` выглядит следующим образом:

```
public void Do()
{
    lock( this )
    {
        for ( int i = 0; i < 4; ++i )
            { Console.Write( " " + i ); Thread.Sleep( 30 ); }
    }
}
```

Для такого варианта метода результат работы программы изменится:

```
Поток Second
```

```
Поток Second
Поток Third
 0 1 2 3 0 1 2 3
```

Асинхронные делегаты

Делегат можно вызвать на выполнение либо синхронно, как во всех приведенных ранее примерах, либо асинхронно с помощью методов `BeginInvoke` и `EndInvoke`.

При вызове делегата с помощью метода `BeginInvoke` среда выполнения создает для исполнения метода отдельный поток и возвращает управление оператору, следующему за вызовом. При этом в исходном потоке можно продолжать вычисления.

Если при вызове `BeginInvoke` был указан метод обратного вызова, этот метод вызывается после завершения потока. Метод обратного вызова также задается с помощью делегата, при этом используется стандартный делегат `AsyncCallback`. В методе обратного вызова для получения возвращаемого значения и выходных параметров применяется метод `EndInvoke`.

Если метод обратного вызова не был указан в параметрах метода `BeginInvoke`, метод `EndInvoke` можно использовать в потоке, инициировавшем запрос.

В листинге 10.8 приводится два примера асинхронного вызова метода, выполняющего разложение числа на множители. Листинг приводится по документации Visual Studio с некоторыми изменениями.

Класс `Factorizer` содержит метод `Factorize`, выполняющий разложение на множители. Этот метод асинхронно вызывается двумя способами: в методе `Num1` метод обратного вызова задается в `BeginInvoke`, в методе `Num2` имеют место ожидание завершения потока и непосредственный вызов `EndInvoke`.

```
using System;
using System.Threading;
```

```
                                        // асинхронный делегат
public delegate bool AsyncDelegate ( int Num, out int m1, out int m2 );

                        // класс, выполняющий разложение числа на множители
public class Factorizer
{
   public bool Factorize( int Num, out int m1, out int m2 )
   {
      m1 = 1;    m2 = Num;
      for ( int i = 2; i < Num; i++ )
         if ( 0 == (Num % i) ) { m1 = i; m2 = Num / i; break; }

      if (1 == m1 ) return false;
      else          return true;
   }
}

                              // класс, получающий делегат и результаты
public class PNum
{
   private int Number;
   public PNum( int number ) { Number = number; }

   [OneWayAttribute()]
                                    // метод, получающий результаты
   public void Res( IAsyncResult ar )
   {
      int m1, m2;
                                  // получение делегата из AsyncResult
      AsyncDelegate ad = (AsyncDelegate)((AsyncResult)ar).AsyncDelegate;

                        // получение результатов выполнения метода Factorize
      ad.EndInvoke( out m1, out m2, ar );
                                           // вывод результатов
      Console.WriteLine( "Первый способ : множители {0} : {1} {2}",
                  Number, m1, m2 );
   }
}
                                        // демонстрационный класс
```

```
                                        // демонстрационный класс
public class Simple
{
                        // способ 1: используется функция обратного вызова
   public void Num1()
   {
      Factorizer     f = new Factorizer();
      AsyncDelegate ad = new AsyncDelegate ( f.Factorize );

      int Num = 1000589023, tmp;
                    // создание экземпляра класса, который будет вызван
                        // после завершения работы метода Factorize
      PNum n = new PNum( Num );

                        // задание делегата метода обратного вызова
      AsyncCallback callback = new AsyncCallback( n.Res );

                             // асинхронный вызов метода Factorize
      IAsyncResult ar = ad.BeginInvoke(
                       Num, out tmp, out tmp, callback, null );
      //
      // здесь - выполнение неких дальнейших действий
      // ...
   }
                  // способ 2: используется ожидание окончания выполнения
   public void Num2()
   {
      Factorizer     f = new Factorizer();
      AsyncDelegate ad = new AsyncDelegate ( f.Factorize );

      int Num = 1000589023, tmp;

                    // создание экземпляра класса, который будет вызван
                        // после завершения работы метода Factorize
      PNum n = new PNum( Num );

                        // задание делегата метода обратного вызова
      AsyncCallback callback = new AsyncCallback( n.Res );
```

```
        IAsyncResult ar = ad.BeginInvoke(
                        Num, out tmp, out tmp, null, null );
                                        // ожидание завершения
        ar.AsyncWaitHandle.WaitOne( 10000, false );

        if ( ar.IsCompleted )
        {
           int m1, m2;
                    // получение результатов выполнения метода Factorize
           ad.EndInvoke( out m1, out m2, ar );
                                        // вывод результатов
           Console.WriteLine( "Второй способ : множители {0} : {1} {2}",
                        Num, m1, m2 );
        }
     }

     public static void Main()
     {
        Simple s = new Simple();
        s.Num1();
        s.Num2();
     }
  }
```

Листинг 10.8. Асинхронные делегаты

Результат работы программы:

```
Первый способ : множители 1000589023 : 7 142941289
Второй способ : множители 1000589023 : 7 142941289
```

Примечание

Атрибут `[OneWayAttribute()]` помечает метод как не имеющий возвращаемого значения и выходных параметров.

На этом данный курс лекций завершается. В него не вошли многие темы, рассмотренные в учебнике [4]: структуры, перечисления, работа с файлами, регулярные выражения, сборки, атрибуты, небезопасный код, основы программирования под Windows. Это связано с формулировкой

основы программирования под Windows. Это связано с формулировкой договора, заключенного автором с издательством ПИТЕР при публикации книги.

Вопросы и задания для самостоятельной работы студента

1. Опишите синтаксис делегата. Как добавить метод в делегат?
2. Расскажите о способах использования делегатов.
3. Какие операции можно выполнять с делегатами?
4. Опишите реализацию событий в C#.
5. Опишите паттерн "наблюдатель".
6. Какой стандартный класс используется для передачи информации о событии?
7. Изучите разделы стандарта C#, касающиеся делегатов.
8. Изучите разделы стандарта C#, касающиеся событий.

Список литературы

1. Бадд Т., Объектно-ориентированное программирование в действии, СПб.: Питер, 1997

2. Биллиг В.А., Основы программирования на C#, М.: Изд-во «Интернет-университет информационных технологий — ИНТУИТ.ру», 2006. — 488 с

3. Гуннерсон Э., Введение в C#. Библиотека программиста, СПб.: Питер, 2001. — 304 с

4. Павловская Т. А., C#. Программирование на языке высокого уровня, URL: http://www.piter.com/book.phtml?978591180174 , СПб.: Питер, 2010. – 432 с

5. Павловская Т. А., C/C++. Программирование на языке высокого уровня, URL: http://www.piter.com/book.phtml?978594723568 , СПб.: Питер, 2010. — 464 с

6. Павловская Т. А., Паскаль. Программирование на языке высокого уровня, СПб.: Питер, 2010. — 464 с

7. Петцольд Ч., Программирование для MS Windows на C#. Том 1, М.: Издательско-торговый дом «Русская Редакция», 2002. — 576 с

8. Петцольд Ч., Программирование для MS Windows на C#. Том 2, М.: Издательско-торговый дом «Русская Редакция», 2002. — 624 с

9. Прайс Д., Гандэрлой М., Visual C#.NET. Полное руководство, Киев: «Век», 2004. — 960 с

10. Секунов Н. , Разработка приложений на C++ и C#. Библиотека программиста, СПб.: Питер, 2003. — 608 с

11. Тай Т., Лэм Х.К., Платформа .NET. Основы, СПб.: Символ-Плюс, 2003. — 336 с

12. Троелсен Э., C# и платформа .NET. Библиотека программиста, СПб.: Питер, 2002. — 796 с

13. Фролов А.В., Фролов Г.В., Язык C#. Самоучитель, М.: Диалог-МИФИ, 2003. — 560 с

14. Шилдт Г., C#: учебный курс, СПб.: Питер, 2002. — 512 с.: ил

15. Конспекты лекций, задания для лабораторных работ, URL: http://ips.ifmo.ru

16. Интегрированная среда разработки, URL: http://msdn.microsoft.com/vstudio/express/visualCsharp/

17. Презентации лекций, список литературы, полезные ссылки, URL: http://pta-ipm.narod.ru

18. Стандарт языка C#, URL: http://www.ecma-international.org/publications/standards/Ecma-334.htm

Содержание