ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

УПРАВЛЕНИЕ ДИСТАНЦИОННОГО ОБУЧЕНИЯ И ПОВЫШЕНИЯ КВАЛИФИКАЦИИ

Кафедра «Автоматизация производственных процессов»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к проведению практических занятий
по дисциплине

«Межплатформенное программирование»

Автор

Шпигун А.В.

Ростов-на-Дону, 2015

Аннотация

Методические указания предназначены для студентов всех форм обучения специальности 230400

Автор

Старший преподаватель каф. «Информационные технологии» Шпигун А.В.

Оглавление

КРАТКАЯ ТЕОРИЯ

Совокупность средств, с помощью которых программы пишутся, корректируются, преобразуются в машинные коды, отлаживаются и запускаются, называют *средой разработки* или *оболочкой. Платформа .Net* или *.Net Framework*– это больше чем просто среда разработки программ, это новое революционное объединение ранее разрозненных технологий компанией Microsoft, которые позволяют разрабатывать разнотипные приложения на различных языках программирования под различные операционные системы.

NET Framework является надстройкой над операционной системой, в качестве которой может выступать любая версия Windows, Unix и вообще любая ОС (по заверению разработчиков), и состоит из ряда компонентов. Так, .NET Framework включает в себя:

1. Четыре официальных языка: C#, VB.NET, Managed C++ и JScript .NET.
2. Общеязыковую объектно-ориентированную среду выполнения CLR (Common Language Runtime), совместно используемую этими языками для создания приложений.
3. Ряд связанных между собой библиотек классов под общим именем FCL (Framework Class Library).

Основным компонентом платформы .NET Framework является общеязыковая среда выполнения программ CLR. Название среды – «общеязыковая среда выполнения» - говорит само за себя: это исполняющая среда, которая подходит для различных языков программирования. К функциям CLR относятся:

1) двухшаговая компиляция: преобразование программы, написанной на одном из языков программирования в управляемый код на промежуточном языке (Microsoft Intermediate Language, MSIL, или просто IL), а затем преобразование IL-кода в машинный код конкретного процессора, который выполняется с помощью виртуальной машины или JIT-компилятора (Just In Time compiler - компилирование точно к нужному моменту);
2) управление кодом: загрузка и выполнение уже готового IL-кода с помощью JIT-компилятора;
3) осуществление доступа к метаданным с целью проверки безопасности кода;
4) управление памятью при размещении объектов с помощью сбор-

щика мусора (Garbage Collector);
5) обработка исключений и исключительных ситуаций, включая межъязыковые исключения;
6) осуществление взаимодействия между управляемым кодом (код, созданный для CLR) и неуправляемым кодом;
7) поддержка сервисов для разработки разнотипных приложений.

Следующим компонентом .Net Framework является FCL – библиотека классов платформы. Эта библиотека разбита на несколько модулей таким образом, что имеется возможность использовать ту или иную ее часть в зависимости от требуемых результатов. Так, например, в одном из модулей содержатся "кирпичики", из которых можно построить Windows-приложения, в другом — "кирпичики", необходимые для организации работы в сети и т.д.

Часть FCL посвящена описанию базисных типов. Тип — это способ представления данных; определение наиболее фундаментальных из них облегчает совместное использование языков программирования с помощью .NET Framework. Все вместе это называется Common Type System (CTS — единая система типов).

Кроме того, библиотека FCL включает в себя Common Language Specification (CLS – общая языковая спецификация), которая устанавливает: основные правила языковой интеграции. Спецификация CLS определяет минимальные требования, предъявляемые к языку платформы .NET. Компиляторы, удовлетворяющие этой спецификации, создают объекты, способные взаимодействовать друг с другом. Поэтому любой язык, соответствующий требованиям CLS, может использовать все возможности библиотеки FCL.

Как уже отмечалось, основными языками, предназначенными для платформы .NET Framework, являются C#, VB.NET, Managed C++ и JScript .NET. Для данных языков Microsoft предлагает собственные компиляторы, переводящие программу в IL-код, который выполняется JIT-компилятором среды CLR. Кроме Microsoft, еще несколько компаний и академических организаций создали свои собственные компиляторы, генерирующие код, работающий в CLR. На сегодняшний момент известны компиляторы для Pascal, Cobol, Lisp, Perl, Prolog и т.д. Это означает, что можно написать программу, например, на языке Pascal, а затем, воспользовавшись соответствующим компилятором, создать управляемый код, который будет работать в среде CLR.

Понятия приложения, проекта, решения

.NET Framework не нала- гает никаких ограничений на воз-

можные типы создаваемых приложений. Тем не менее, давайте рассмотрим некоторые наиболее часто встречающиеся типы приложений:

1) Консольные приложения позволяют выполнять вывод на «консоль», то есть в окно командного процессора.
2) Windows-приложения, использующие элементы интерфейса Windows, включая формы, кнопки, флажки и т.д.
3) Web-приложения представляют собой web-страницы, которые могут просматриваться любым web-браузером.
4) Web-сервисы представляют собой распределенные приложения, которые позволяют обмениваться по Интернету практически любыми данными с использованием единого синтаксиса независимо от того, какой язык программирования применялся при создании web-службы и на какой системы она размещена.

Приложение, находящееся в процессе разработки, называется проектом. Несколько приложений могут быть объединены в решение (solution).

Удобной средой разработки приложений является Visual Studio .Net.

Компиляция и выполнение программы в среде CLR

В прошлом почти все компиляторы генерировали код для конкретных процессорных архитектур. Все CLR-совместимые компиляторы вместо этого генерируют IL-код, который также называется управляемым модулем, потому что CLR управляет его жизненным циклом и выполнением. Рассмотрим составные части управляемого модуля:

1. *Заголовок PE32 или PE32+:* Файл с заголовком в формате PE32 может выполняться в 32- или 64-разрядной ОС, а с заголовком PE32+ только в 64-разрядной ОС. Заголовок показывает тип файла: GUI, GUI или DLL, он также имеет временную метку, показывающую, когда файл был собран. Для модулей, содержащих только IL-код, основной объем информации в PE-заголовке игнорируется, Для модулей, содержащих процессорный код, этот заголовок содержит сведения о процессорном коде.
2. *Заголовок CLR:* Содержит информацию, которая превращает этот модуль в управляемый. Заголовок включает нужную версию CLR, некоторые флаги, метку метаданных, точки входа в управляемый модуль (метод Main), месторасположение и размер метаданных модуля, ресурсов и т.д.
3. *Метаданные* - это набор таблиц данных, описывающих то, что

определено в модуле. Есть два основных вида таблиц: описывающие типы и члены, определенные в вашем исходном коде, и описывающие типы и члены, на которые имеются ссылки в вашем исходном коде. Метаданные служат многим целям:

a. устраняют необходимость в заголовочных и библиотечных файлах при компиляции, так как все сведения о типах и членах, на которые есть ссылки, содержатся в файле с IL-кодом, в котором они реализованы. Компиляторы могут читать метаданные прямо из управляемых модулей.
b. при компиляции IL-кода в машинный код CLR выполняет верификацию (проверку «безопасности» выполнения кода) используя метаданные, например, нужное ли число параметров передается методу, корректны ли их типы, правильно ли используется возвращаемое значение и т.д.
c. позволяют сборщику мусора отслеживать жизненный цикл объектов и т.д.

4. IL-код: управляемый код, создаваемый компилятором при компиляции исходного кода. Во время исполнения CLR компилирует IL-код в команды процессора.

По умолчанию CLR-совместимые компиляторы генерируют управляемый код, безопасность выполнения которого поддается проверке средой CLR. Вместе с тем возможно разрабатывать неуправляемый или «небезопасный» код, которому разрешается работать непосредственно с адресами памяти и управлять байтами в этих адресах. Эта возможность, обычно полезна при взаимодействии с неуправляемым кодом или при необходимости добиться максимальной производительности при выполнении критически важных алгоритмов. Однако использовать неуправляемый код довольно рискованно, т.к. он способен разрушить существующие структуры данных.

Чтобы понять принцип выполнения программы в среде CLR рассмотрим небольшой пример:

Управляемый модуль

```
static void Main ()
{
  Console.WriteLine("УРА!");
  Console.WriteLine(··);
}
```

```
Console
...
Static                    void
WriteLine(string);
...
```

JITCompiler

1. В сборке, реализующей данный тип (Console), найти в метаданных вызываемый метод (WriteLine).
2. Взять из метаданных IL-код для этого метода.
3. Выделить блок памяти.
4. Скомпилировать IL-кода команды процессора и сохранить процессорный код в памяти, выделенной на этапе 3.
5. Изменить точку входа метода в таблице типа, чтобы она указывала на блок памяти, выделенный на этапе 3.
6. Передать управление процессорному коду, содержащемуся в выделенном блоке памяти.

Непосредственно перед исполнением функции Main CLR находит все типы, на которые ссылается ее код. В нашем случае метод Main ссылается на единственный тип — Console, и CLR выделяет единственную внутреннюю структуру WriteLine.

Когда Main первый раз обращается к WriteLine, вызывается функция JITCompiler (условное название), которая отвечает за компиляцию IL-кода вызываемого метода в собственные команды процессора. Функции JITCompiler известен вызываемый метод и тип, в котором он определен. JITCompiler ищет в метаданных соответствующей сборки IL-код вызываемого метода, затем проверяет и компилирует IL-код в собственные команды процессора, которые сохраняются в динамически выделенном блоке памяти. После этого JITCompiler возвращается к внутренней структуре данных типа и заменяет адрес вызываемого метода адресом блока памяти, содержащего собственные команды процессора. В завершение JITCompiler передает управление коду в этом блоке памяти. Далее управление возвращается в Main, который продолжает работу в обычном порядке.

Затем Main обращается к WriteLine вторично. К этому моменту код WriteLine уже проверен и скомпилирован, так что производится обращение к блоку памяти, минуя вызов JITCompiler. Отработав, метод WriteLine возвращает управление Main.

Таким образом, за счет такой компиляции производительность теряется только при первом вызове метода. Все последующие обращения к одной и той же структуре выполняются «на полной скорости», без повторной верификация и компиляция.

ЛАБОРАТОРНАЯ РАБОТА №1

Знакомство со средой разработки Visual Studio. Разработка простых консольных приложений.

Задание №1 (Среда разработки, Структура каталогов проекта)

Можно создавать файлы с исходным кодом на C# с помощью обычного текстового редактора (например, Блокнота) и компилировать их в управляемые модули с помощью компилятора командной строки, который является составной частью .NET Framework. Однако наиболее удобно для этих целей использовать VS, потому что:

1. VS автоматически выполняет все шаги, необходимые для компиляции исходного кода.

2. Текстовый редактор VS настроен для работы с теми языками, которые поддерживаются VS, например C#, поэтому он может интеллектуально обнаруживать ошибки и подсказывать в процессе ввода, какой именно код необходим.

3. В состав VS входят программы, позволяющие создавать Windows- и Web-приложения путем простого перетаскивания мышью элементов пользовательского интерфейса.

4. Многие типы проектов, создание которых возможно на C#, могут разрабатываться на основе "каркасного" кода, заранее включаемого в программу. Вместо того чтобы каждый раз начинать с нуля, VS позволяет использовать уже имеющиеся файлы с исходным кодом, что уменьшает временные затраты на создание проекта.

Создание проекта

Для создания проекта следует запустить VS, а затем в главном меню VS выбрать команду ***File – New - Project*** . После чего откроется диалоговое меню ***New Project*** (см. рис.1).

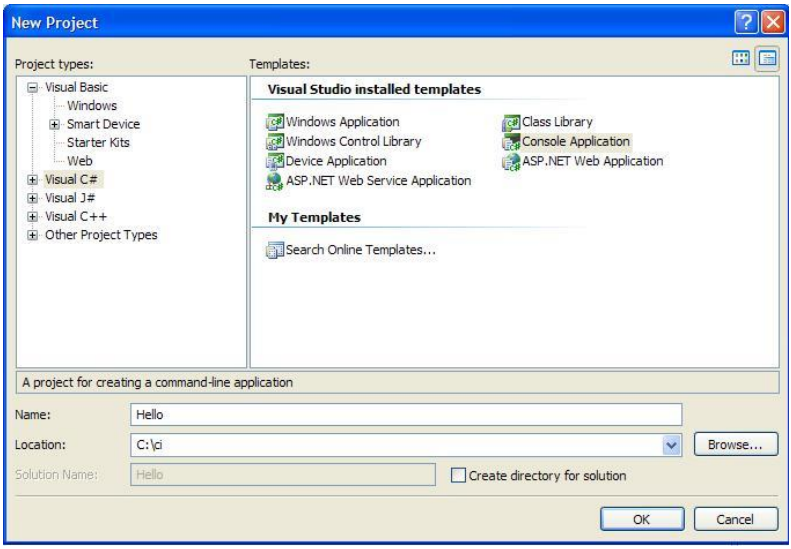


Рис.1.

В поле ***Project types*** следует выбрать *Visual C#*, в поле ***Templates*** – *Console Application*.

В строчке ***Name*** введите имя приложения *Hello*. Обратите внимание на то, что это же имя появится в строчке ***Solution Name***. Уберите галочку в поле ***Create directory for Application*** (пока мы создаем простое приложение, и нам нет необходимости усложнять его структуру).

В строке ***Location*** определите положение на диске, куда нужно сохранять ваш проект. И нажмите кнопку ***OK***. Примерный вид экрана изображен на рис 2.

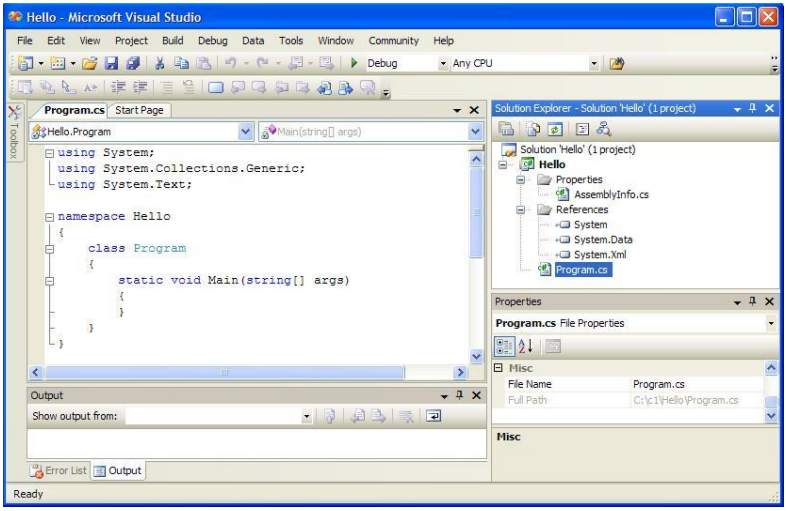


Рис. 2.

В правой верхней части располагается окно управления проектом ***Solution Explorer***. Если оно закрыто, то его можно включить командой ***View - Solution Explorer***. В этом окне перечислены все ресурсы, входящие в проект:

1) AssemblyInfo.cs – информация о *сборке*.

   Компилятор в качестве результата своего выполнения создает так называемую *сборку* – файл с расширением exe или dll, который содержит IL-код и метаданные. (см. и слушай лекции)

2) System, System.Data, System.Xml – ссылки на стандартные библиотеки.
3) Program.cs - текст программы на языке C#.

Замечание. В других версиях VS сюда же включается файл с расширением ico, отвечающий за вид ярлыка приложения.

В правой нижней части экрана располагается окно свойств ***Properties***. Если оно закрыто, то его можно включить командой ***View - Properties***. В этом окне отображаются важнейшие характеристики выделенного элемента.

Основное пространство экрана занимает окно редактора, в котором располагается текст программы, созданный средой автоматически. Текст представляет собой каркас, в который программист будет добавлять нужный код. При этом зарезервированные слова отображаются синим цветом, комментарии – зеленым, основной текст – черным.

Текст структурирован. Щелкнув на знак минус, мы скроем блок кода, щелкнув на знаке плюс – откроем.

Откроем папку, содержащую проект, и рассмотрим ее структуру (см. рис.3). Файлы, выделенные жирным шрифтом, появятся только после компиляции.

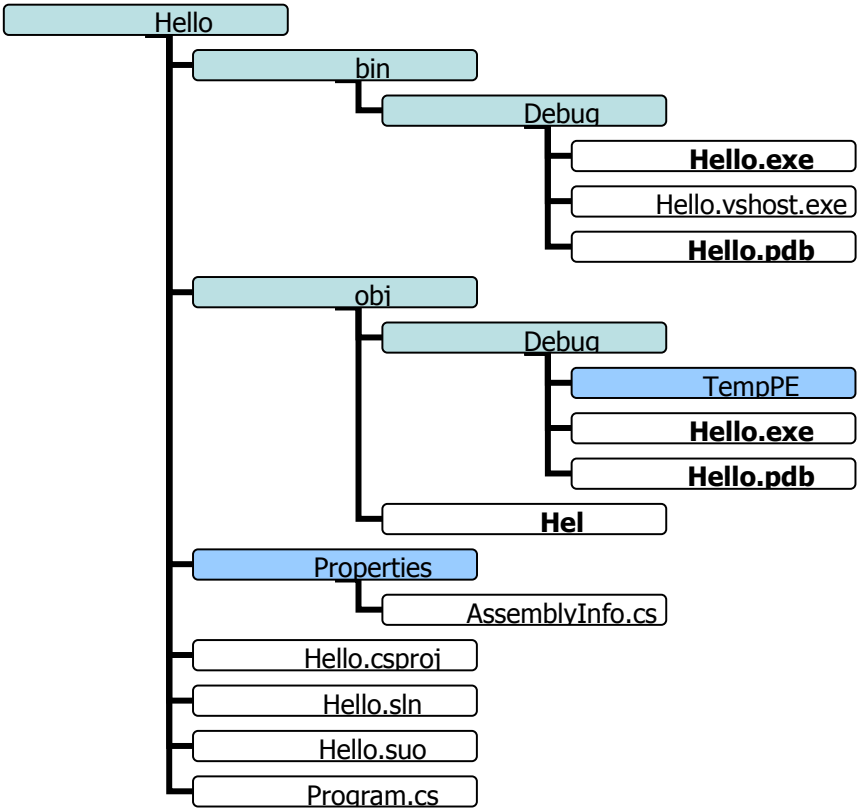


Рис. 3.

На данном этапе особый интерес для нас будут представлять следующие файлы:

1. *Hello.sln* – основной файл, отвечающий за весь проект. Если необходимо открыть проект для редактирования, то нужно выбрать именно этот файл. Остальные файлы откроются автоматически.
2. *Program.cs* – файл, в котором содержится исходный код - код, написанный на языке C#. Именно с этим файлом мы и будем непосредственно работать.
3. *Hello.exe* – файл, в котором содержатся сгенерированный IL-код и метаданные проекта. Другими словами, этот файл и есть готовое

приложение, которое может выполняться на любом компьютере, на котором установлена платформа .Net.

Теперь рассмотрим сам текст программы.

using System – это директива, которая разрешает использовать имена стандартных классов из пространства имен ***System*** непосредственно без указания имени пространства, в котором они были определены.

Ключевое слово ***namespace*** создает для проекта свое собственное пространство имен, которое по умолчанию называется именем проекта. В нашем случае пространство имен называется Hello. Однако программист вправе указать другое имя. Пространство имен ограничивает область применения имен, делая его осмысленным только в рамках данного пространства. Это сделано для того, чтобы можно было давать имена программным объектам, не заботясь о том, что они совпадут с именами в других приложениях. Таким образом, пространства имен позволяют избегать конфликта имен программных объектов, что особенно важно при взаимодействии приложений.

C# - объектно-ориентированный язык, поэтому написанная на нем программа будет представлять собой совокупность взаимодействующих между собой классов. Автоматически был создан класс с именем ***Program*** (в других версиях среды может создаваться класс с именем ***Class1***).

Данный класс содержит только один метод - метод ***Main()***. Метод Main() является точкой входа в программу, т.е. именно с данного метода начнется выполнение приложения. Каждая программа на языке C# должна иметь метод Main ().

Замечание. Технически возможно иметь несколько методов Main() в одной программе, в этом случае потребуется с помощью параметра командной строки сообщить компилятору C#, какой именно метод Main() является точкой входа в программу.

Метод Main() имеет одну важную особенность. Перед объявлением типа возвращаемого значения ***void*** (который означает, что метод не возвращает значение) стоит ключевое слово ***static***, которое означает что метод Main() можно вызывать, не создавая объект типа ***Program***.

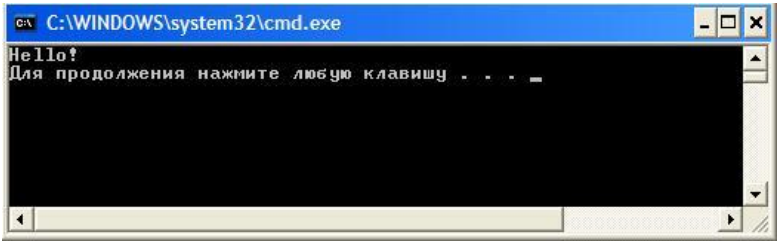
Замечание. В некоторых версиях требуется, чтобы перед словом ***static*** стояло слово ***public***.

Добавим в метод следующий код: ***Console.WriteLine("Hello!");***

Здесь *Console* имя стандартного класса из пространства имен System. Его метод *WriteLine* выводит на экран текст, заданный в кавычках

Для запуска программы следует нажать клавишу F5 или выпол-

нить команду ***Debug-Start Debugging.*** Если программа выполнена без ошибок, то сообщение выведется в консольное окно, которое мелькнет и быстро закроется. Чтобы просмотреть сообщение в нормальном режиме нужно нажать клавиши Ctrl+F5 или выполнить команду ***Debug-Start Without Debugging.*** В нашем случае откроется следующее консольное окно:



Если код программы будет содержать ошибки, например, пропущена точка с запятой после команды вывода, то после нажатия клавиши F5 откроется диалоговое окно, в котором выведется сообщение о том, что обнаружена ошибка, и вопрос, продолжать ли работу дальше. Если вы ответите ***Yes***, то будет выполнена предыдущая удачно скомпилированная версия программы. Иначе процесс будет остановлен и управление передано окну списка ошибок ***Error List***.

Задание 2. Переменные и константы

Переменная представляет собой типизированную область памяти. Программист создает переменную, объявляя ее тип и указывая имя. При объявлении переменной ее можно инициализировать (присвоить ей начальное значение), а затем в любой момент ей можно присвоить новое значение, которое заменит собой предыдущее.

Добавьте следующий код в функцию main. Проверьте его работу.

```
static void Main()
    {
      int i=10;//объявление и инициализация целочисленной переменной i
      Console.WriteLine(i);      //просмотр значения переменной
      i=100;                     //изменение значение переменной
      Console.WriteLine(i);
    }
```

В языке C# требуется, чтобы переменные были явно проинициализированы до их использования. Проверим этот факт на примере.

Закомментируйте предыдущий код и введите следующий. Проверьте результат работы. Зафиксируйте полученные ре-

зультаты.

```
static void Main()
    {
      int i;
      Console.WriteLine(i);
    }
```

При попытке скомпилировать этот пример в списке ошибок будет выведено следующее сообщение: *Use of unassigned local variable 'i'* (используется неинициализированная локальная переменная i).

Однако инициализировать каждую переменную **необязательно, но необходимо присвоить ей значение до того, как она будет использована.**

Константа - это переменная, значение которой нельзя изменить. Константы бывают трех видов: *литералы, символические константы* и *перечисления.*

В операторе присваивания: x=32;

число 32 является литеральной константой. Его значение всегда равно 32 и его нельзя изменить. (x – является переменной и его значение изменить можно!!!!)

Символические константы именуют постоянные значения. Определение символической константы происходит следующим образом:

const <тип> <идентификатор> = <значение>;

Рассмотрим пример: (закомментируйте предыдущий код функции main, заменив его следующим кодом)

```
static void Main()
    {
      const int i=10; //объявление целочисленной константы i
      Console.WriteLine(i);          //просмотр значения константы
      i=100;                  //ошибка
      Console.WriteLine(i);
    }
```

Задание. Измените программу так, чтобы при объявлении константы не происходила инициализация. Как на это отреагирует компилятор и почему?

Перечисления (enumerations) являются альтернативой константам. Перечисление - это особый размерный тип, состоящий из набора именованных констант (называемых *списком перечисления).*

Синтаксис определения перечисления следующий:

[атрибуты] [модификаторы] enum <имя> [: базовый тип] {список-перечисления констант(через запятую)};

Замечание. Атрибуты и модификаторы, являются необязательными элементами этой конструкции. Более подробно мы рассмотрим их позже.

Базовый тип - это тип самого перечисления. Если не указать базовый тип, то по умолчанию будет использован тип int. В качестве базового типа можно выбрать любой целый тип, кроме char. Пример использования перечисления:

Задание. Дополните свою программу следующим кодом и обеспечьте его работоспособность.

```
class Program
  {
    enum gradus:int
    {
      min=0,
      krit=72,
      max=100,
    }
    static void Main()
    {
      Console.WriteLine("минимальная температура=" + (int) gradus.min);
      Console.WriteLine("критическая температура=" + (int)gradus.krit);
      Console.WriteLine("максимальная температура=" + (int)gradus.max);
    }
  }
```

Замечания

1. Запись (int) gradus.min используется для явного преобразования перечисления к целому типу. Если убрать (int), то на экран будет выводиться название констант.
2. Символ + в записи "минимальная температура=" + (int) gradus.min при обращении к методу WriteLine означает, что строка "минимальная температура=" будет «склеена» со строковым предствлением значения (int) gradus.min. В результате получится новая строка, которая и будет выведена на экран.

Задание3. Организация ввода-вывода данных. Форматирование.

Программа при вводе данных и выводе результатов взаимодействует с внешними устройствами. Совокупность стандартных устройств ввода (клавиатура) и вывода (экран) называется консолью. В

языке C# нет операторов ввода и вывода. Вместо них для обмена данными с внешними устройствами используются специальные объекты. В частности, для работы с консолью используется стандартный класс Console, определенный в пространстве имен System.

Вывод данных

В приведенных выше примерах мы уже рассматривали метод WriteLine, реализованный в классе Console, который позволяет организовывать вывод данных на экран. Однако существует несколько способов применения данного метода:

1. Console.WriteLine(x); //на экран выводится значение идентификатора x
2. Console.WriteLine("x=" + x +"y=" + y); /* на экран выводится строка, образованная последовательным слиянием строки "x=", значения x, строки "y=" и значения y */
3. Console.WriteLine("x={0} y={1}", x, y); /* на экран выводится строка, формат которой задан первым аргументом метода, при этом вместо параметра {0} выводится значение x, а вместо {1} – значение y*/

Замечание. Рассмотрим следующий фрагмент программы:

```
int i=3, j=4;
Console.WriteLine("{0} {1}", i, j);
```

При обращении к методу WriteLine через запятую перечисляются три аргумента: "{0} {1}", i, j. Первый аргумент определяет формат выходной строки. Следующие аргументы нумеруются с нуля, так переменная i имеет номер 0, j – номер 1. Значение переменной i будет помещено в выходную строку на место {0}, а значение переменной j –на место {1}. В результате на экран будет выведена строка: 3 4. Если мы обратимся к методу WriteLine следующим образом Console.WriteLine("{0} {1} {2)", j, i, j), то на экран будет выведена строка: 4 3 4.

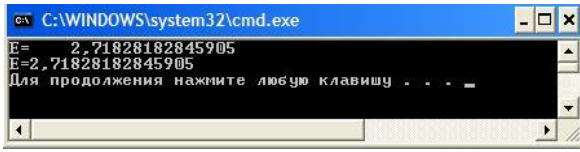
Последний вариант использования метода WriteLine является наиболее универсальным, потому что он позволяет не только выводить данные на экран, но и управлять форматом их вывода. Рассмотрим несколько примеров:

1) *Использование управляющих последовательностей:*

Управляющей последовательностью называют определенный символ, предваряемый обратной косой чертой. Данная совокупность символов интерпретируется как одиночный символ и используется для представления кодов символов, не

имеющих графического обозначения (например, символа перевода курсора на новую строку) или символов, имеющих

```
static void Main()
  {
   double x= Math.E;
   Con-
sole.WriteLine("E={0,20}", x);
   Con-
sole.WriteLine("E={0,10}", x);
   }
```



специальное обозначение в символьных и строковых константах (например, апостроф). Рассмотрим управляющие символы:

Вид	**Наименование**
\a	Звуковой сигнал
\b	Возврат на шаг назад
\f	Перевод страницы
\n	Перевод строки
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\\	Обратная косая черта
\’	Апостроф
\”	Кавычки

Пример:

```
static void Main()
  {
   Console.WriteLine("Ура!\nСегодня \"понедельник\"!!!");
  }
```

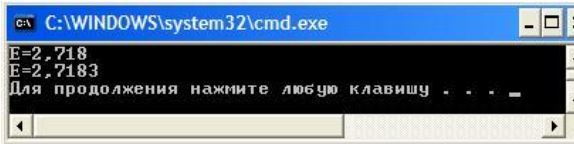
2) *Управление размером поля вывода:*

Первым аргументом WriteLine указывается строка вида {n, m} – где n определяет номер идентификатора из списка аргументов метода WriteLine, а m – количество позиций (размер поля вывода), отводимых под значение данного идентификатора. При этом значение идентификатора выравнивается по правому краю. Если выделенных позиций для размещения значения идентификатора окажется недостаточно, то автоматически добавиться необходимое количество позиций. Пример:

3) Управление размещением вещественных данных:

Первым аргументом WriteLine указывается строка вида {n: ##.###} – где n определяет номер идентификатора из списка аргументов метода WriteLine, а ##.### определяет формат вывода вещественного числа. В данном случае под целую часть числа отводится две позиции, под дробную – три. Если выделенных позиций для размещения целой части значения идентификатора окажется недостаточно, то автоматически добавитьcя необходимое количество позиций. Пример:

```
static void Main()
  {
   double x= Math.E;
   Con-
sole.WriteLine("E={0:##.###
}", x);
   Con-
sole.WriteLine("E={0:.####}"
, x);
  }
```



Задание. Измените программу так, чтобы число e выводилось на экран с точностью до 6 знаков после запятой.

4) Управление форматом числовых данных:

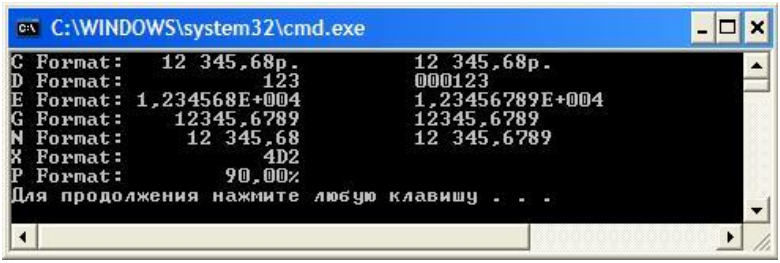
Первым аргументом WriteLine указывается строка вида {n: <спецификатор>m} – где n определяет номер идентификатора из списка аргументов метода WriteLine, <спецификатор> - определяет формат данных, а m – количество позиций для дробной части значения идентификатора. В качестве спецификаторов могут использоваться следующие значения:

Параметр	Формат	Значение
C или c	Денежный. По умолчанию ставит знак р. Изменить его можно с помощь объекта NumberFormatInfo	Задается количество десятичных разрядов.
D или d	Целочисленный (используется только с целыми числами)	Задается минимальное количество цифр. При необходимости результат дополняется начальными нулями
E или e	Экспоненциальное представление чисел	Задается количество символов после запятой. По умолчанию используется 6

F или f	Представление чисел с фиксированной точкой	Задается количество символов после запятой
G или g	Общий формат (или экспоненциальный, или с фиксированной точкой)	Задается количество символов после запятой. По умолчанию выводится целая часть
N или n	Стандартное форматирование с использованием запятых и пробелов в качестве разделителей между разрядами	Задается количество символов после запятой. По умолчанию – 2, если число целое, то ставятся нули
X или x	Шестнадцатеричный формат	
P или p	Процентный	

Пример:

```
static void Main()
  {
   Console.WriteLine("C Format:{0,14:C} \t{0:C2}", 12345.678);
   Console.WriteLine("D Format:{0,14:D} \t{0:D6}", 123);
   Console.WriteLine("E Format:{0,14:E} \t{0:E8}", 12345.6789);
   Console.WriteLine("G Format:{0,14:G} \t{0:G10}", 12345.6789);
   Console.WriteLine("N Format:{0,14:N} \t{0:N4}", 12345.6789);
   Console.WriteLine("X Format:{0,14:X} ", 1234);
   Console.WriteLine("P Format:{0,14:P} ", 0.9);
  }
```



Ввод данных

Для ввода данных обычно используется метод ReadLine, реализованный в классе Console. Особенностью данного метода является то, что в качестве результата он возвращает строку (string). Пример:

```
static void Main()
 {
  string s = Console.ReadLine();
  Console.WriteLine(s);
 }
```

Для того чтобы получить числовое значение необходимо воспользоваться преобразованием данных. Пример:

```
static void Main()
 {
  string s = Console.ReadLine();
  int x = int.Parse(s); //преобразование строки в число
  Console.WriteLine(x);
 }
```

Или сокращенный вариант:

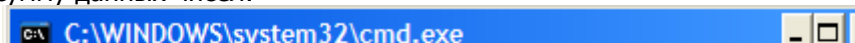
```
static void Main()
 {
  int x = int.Parse(Console.ReadLine()); //преобразование введенной строки в
число
  Console.WriteLine(x);
 }
```

Для преобразования строкового представления целого числа в тип int мы используем метод int.Parse(), который реализован для всех числовых типов данных. Таким образом, если нам потребуется преобразовать строковое представление в вещественное, мы можем воспользоваться методом float.Parse() или double.Parse(). В случае, если соответствующее преобразование выполнить невозможно, то выполнение программы прерывается и генерируется исключение System.FormatExeption (входная строка имела неверный формат).

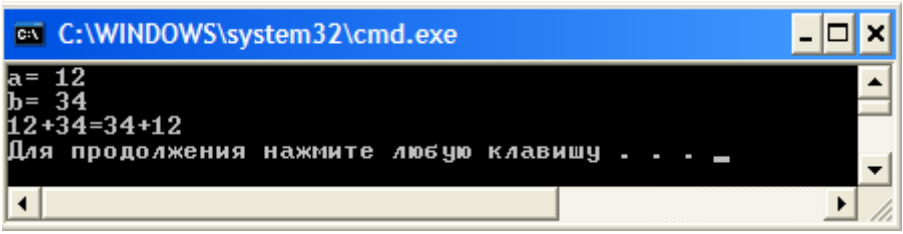
Задание. Измените предыдущий фрагмент программы так, чтобы с клавиатуры вводилось вещественное число, а на экран это число выводилось с точностью до 3 знаков после запятой.

Написать программу, которая, реализует диалог с пользователем:

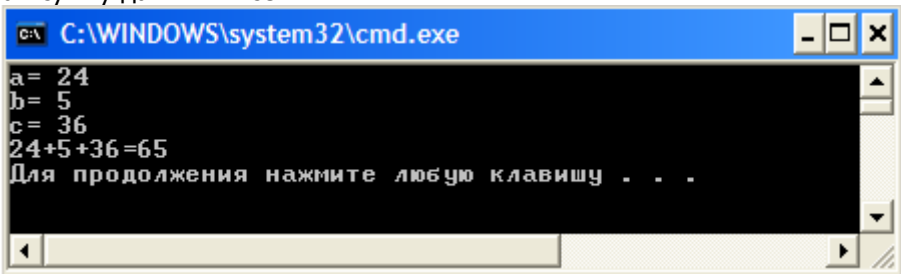
1) запрашивает с клавиатуры два целых числа, и выводит на экран сумму данных чисел:



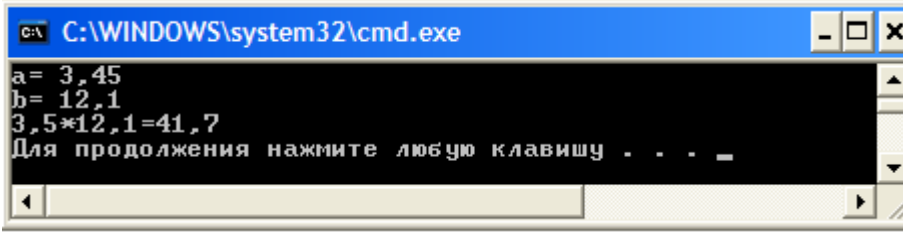
2) запрашивает с клавиатуры три целых числа, и выводит на экран сумму данных чисел в прямом и обратном порядке:



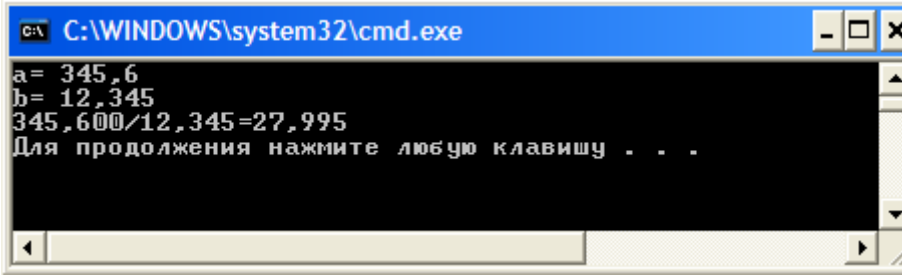
3) запрашивает с клавиатуры три целых числа, и выводит на экран сумму данных чисел:



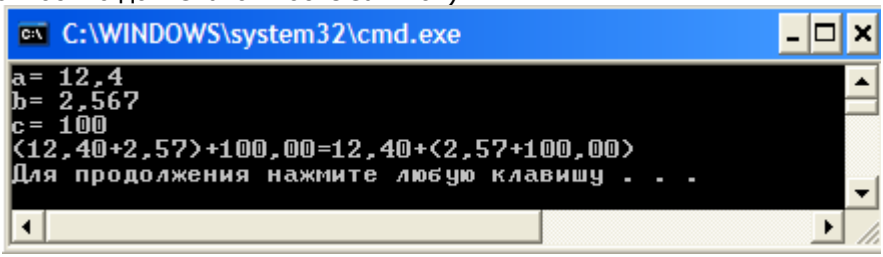
4) запрашивает с клавиатуры два вещественных числа, и выводит на экран произведение данных чисел (вещественные числа выводятся с точностью до 1 знака после запятой):



5) запрашивает с клавиатуры два вещественных числа, и выводит на экран результат деления первого числа на второе (вещественные числа выводятся с точностью до 3 знаков после запятой):



6) запрашивает с клавиатуры три вещественных числа, и выводит на следующее сообщение (вещественные числа выводятся с точностью до 2 знаков после запятой):



Задание 4. Самостоятельно.

Используя Интернет и дополнительную литературу найти ответы на следующие вопросы:

1. Чем отличается метод Console.WriteLine() от метода Console.Write()?
2. Чем отличается метод Console.ReadLine() от метода Console.Read()?
3. Какой тип имеет литеральная константа 3.2?
4. Как явным образом уточнить тип литеральной константы?
5. Что обозначается константой NaN? И в каких случаях компилятором используется данная константа?

Вопросы к лабораторной работе

1. Какие существуют преимущества использования комплекса VS?
2. Как создать новый проект? Опишите структуру каталогов проекта.
3. Какие файлы создаются автоматически? Какой файл отвечает за весь проект? В каком сохраняется программа?
4. Какое ключевое слово используется для обозначения про-

странства имён? Для чего применяется пространство имён?
5. Для каких целей используется директива using?
6. Какие спецификаторы должны быть у метода main? Почему?
7. Чем отличается запуск программы по кнопке F5 и Ctrl+F5?
8. Какие ограничения накладываются на переменные в языке C#?
9. Как производится описание константы? Классификации констант? Типы констант?
10. Почему при использовании перечисления необходимо использовать приведение типа?
11. Как создать перечисление? Какие типы данных могут быть базовыми для перечисления?
12. Способы форматирования данных. Примеры использования. Управляющие последовательности.
13. Ввод данных. Примеры приведения типов с использование Parse.

ЛАБОРАТОРНАЯ РАБОТА №2.

Тема: Разработка библиотек dll на c#. Операции и операторы языка C#

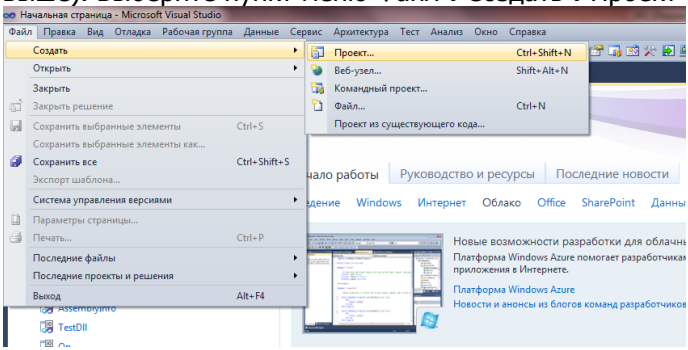
Внимание! Рекомендации о порядке защиты лабораторной работы указаны в конце текста!

Часть 1. Разработка dll на C#

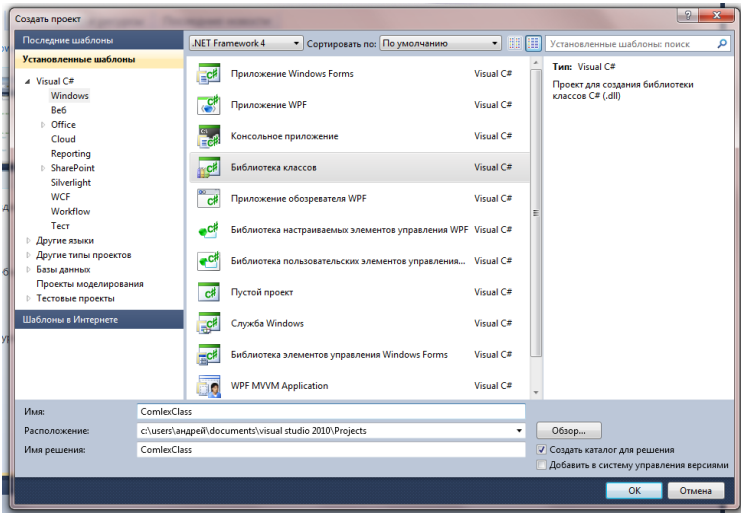
Задание: Разработать библиотеку dll для работы с ком-

плексными числами и единственной операцией сложения.

1. Создание библиотеки dll с помощью Visual Studio (2010 и выше). Выберите пункт меню Файл→Создать→Проект



2. Выберите проект «Библиотека классов». (Расширение dll). Укажите Имя проекта и расположение проекта.

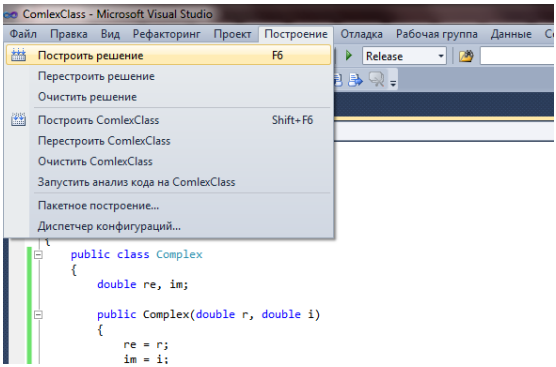


3. Введите текст класса приведенного класса. В обучающих целях ряд ключевых и воспроизводимых из контекста слов заменены «……»

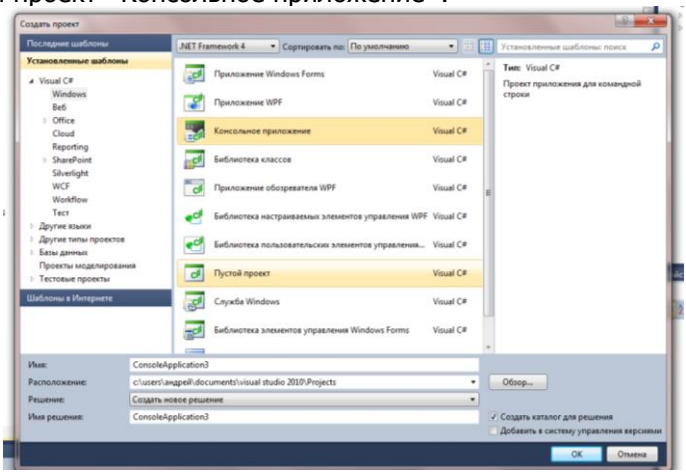
```
….. System;


….. ComplexClass
{
    ….. class Complex
    {
        double re, im;

        public Complex(….. r, ….. i)
        {
            re = r;
            im = i;
        }


        public static ….. sum(Complex C1, Complex C2)
        {
            return  new  …..(C1.re  +  C2.re,  C1.im  +
C2.im);
        }

        public ….. ….. ToString()
        {
            ….. ""+re+"  "+im;
        }

    }
}
```

---

4. Устанавливаем конфигурацию решения в состояние «Release». Выбираем пункт меню Построение→Построить решение:



5. Для тестирования созданной dll библиотеки добавляем новый проект «Консольное приложение»:

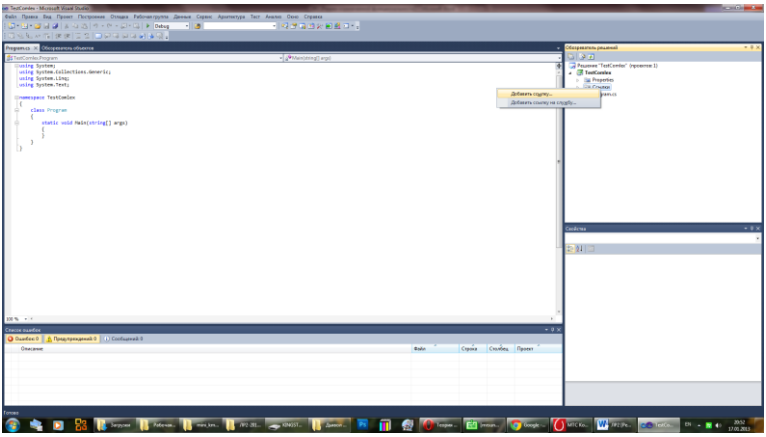


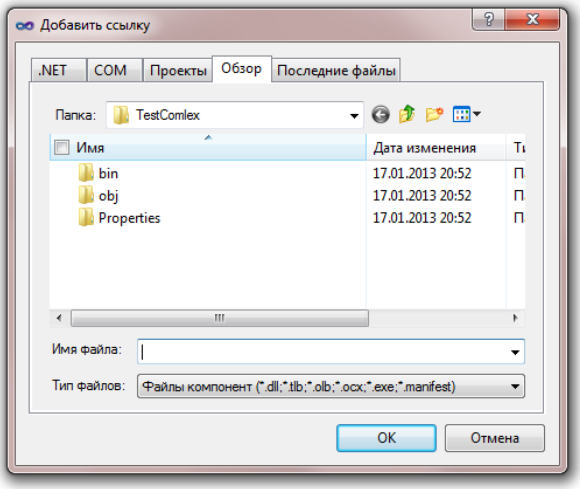
6. Введите текст класса приведенного класса. В обучающих целях ряд ключевых и воспроизводимых из контекста слов заменены «.....»

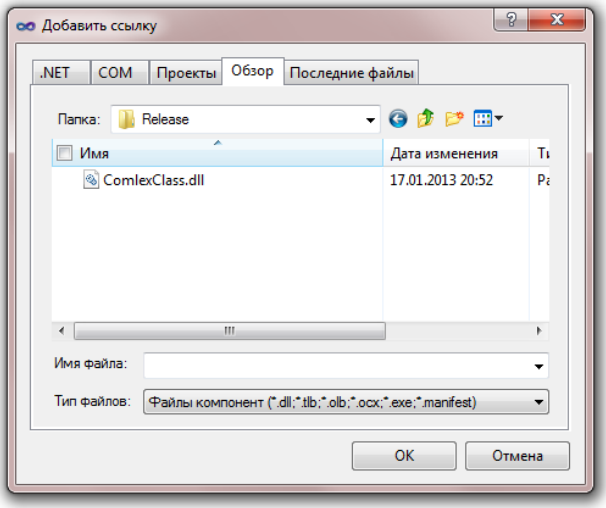
```
using System;
….. ComplexClass;
```

```
namespace TestComlex
{
    ….. Program
    {
        static void Main(string[] args)
        {
            Complex t1 = new …..(1,1);
            ….. t2 = new Complex(2,3),t3;
            t3 = Complex……..(t1,t2);
            …..WriteLine(t1);
            …..WriteLine(t2);
            ……..(t3);
        }
    }
}
```

7. Подключаем к проекту созданную ранее библиотеку. Для этого необходимо указать ссылку на файл построенный на шаге 4.







8. Постройте проект и протестируйте его работу. Убедитесь, что класс и функции описанные во внешней библиотеке функционирую нормально.

Часть 2. Операции языка C#

Полный список операций языка C# в соответствии с их приоритетами (по убыванию приоритетов, операции с разными приоритетами разделены чертой) приведен в следующей таблице:

Операция	Описание
.	Доступ к элементу
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
+	Унарный плюс
-	Арифметическое отрицание
!	Логическое отрицание
~	Поразрядное отрицание
++x	Префиксный инкремент
--x	Префиксный декремент
(тип) x	Преобразование типа
*	Умножение
/	Деление
%	Остаток от деления
<<	Сдвиг влево
>>	Сдвиг вправо
<	Меньше
>	Больше
<=	Меньше или равно
>=	Больше или равно
is	Проверка принадлежности типу
as	Приведение типа
==	Равно
!=	Не равно
&	Поразрядное И
^	Поразрядное исключающее ИЛИ
|	Поразрядное ИЛИ
&&	Логическое И

||	Логическое ИЛИ
? :	Условная операция
=	Простое 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

В данном разделе мы подробно рассмотрим только часть операций, остальные операции будут вводиться по мере необходимости. Если после изучения других языков операции покажутся Вам знакомыми, то соответствующие из них можно пропустить. Но! Внимание!! Вы должны быть уверены, что сможете объяснить те явления, которые описываются в соответствующем задании!!!

Замечание. Операции можно классифицировать по количеству операндов на: унарные – воздействуют на один операнд, бинарные – воздействуют на два операнда, тернарные – воздействует на три операнда. Некоторые символы используются для обозначения как унарных, так и бинарных операций. Например, символ «минус» используется как для обозначения унарной операции – арифметического отрицания, так и для обозначения бинарной операции вычитание. Будет ли данный символ обозначать унарную или бинарную операцию, определяется контекстом, в котором он используется.

1. Инкремент (++) и декримент(--).

Эти операции имеют две формы записи — префиксную, когда операция записывается перед операндом, и

постфиксную – операция записывается после операнда. Префиксная операция инкремента (декремента) увеличивает (уменьшает) свой операнд и возвращает измененное значение как результат. Постфиксные версии инкремента и декремента возвращают первоначальное значение операнда, а затем изменяют его.

Рассмотрим эти операции на примере.

```
static void Main()
    {
        int i = 3, j = 4;
        Console.WriteLine(«{0} {1}», i, j);
        Console.WriteLine(«{0} {1}», ++i, --j);
        Console.WriteLine(«{0} {1}», i++, j--);
        Console.WriteLine(«{0} {1}», i, j);
    }
```

Результат работы

```
3 4
4 3
4 3
5 2
```

Замечание. Префиксная версия требует существенно меньше действий: она изменяет значение переменной и запоминает результат в ту же переменную. Постфиксная операция должна отдельно сохранить исходное значение, чтобы затем вернуть его как результат. Для сложных типов подобные дополнительные действия могут оказаться трудоемки. Поэтому постфиксную форму имеет смысл использовать только при необходимости.

2. Операция new.
   Используется для создания нового объекта. С помощью ее можно создавать как объекты ссылочного типа, так и размерные, например:

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

3. Отрицание:
   Арифметическое отрицание (-) – меняет знак операнда

на противоположный.

Логическое отрицание (!) – определяет операцию инверсия для логического типа.

Рассмотрим эти операции на примере.

```
static void Main()                                  Результат работы программы:
      {
         int i = 3, j=-4;
         bool a = true, b=false;
         Console.WriteLine(«{0} {1}», -i, -j);              -3   4
         Console.WriteLine(«{0} {1}», !a, !b);              False True
      }
```

4. Явное преобразование типа.
   Используется для явного преобразования из одного типа в другой. Формат операции:

(тип) выражение;

Рассмотрим эту операцию на примере.

```
static void Main()                                                  Результат рабо-
      {                                                          ты программы:
         int i = -4;
         byte j = 4;
         int a = (int)j;  //преобразование без потери точности
         byte b = (byte)i; //преобразование с потерей точности
         Console.WriteLine(«{0} {1}», a, b);                          4   252
      }
```

5. Умножение (*), деление (/) и деление с остатком (%).
   Операции умножения и деления применимы для целочисленных и вещественных типов данных. Для других типов эти операции применимы, если для них возможно неявное преобразование к целым или вещественным типам. При этом тип результата равен «наибольшему» из типов операндов, но не менее int. Если оба операнда при делении целочисленные, то и результат тоже целочисленный.

Рассмотрим эти операции на примере.

```
static void Main()
```

```
        {
int i = 100, j = 15;
double a = 14.2, b = 3.5;
Console.WriteLine(«{0} {1} {2}», i*j, i/j, i%j);
Console.WriteLine(«{0} {1} {2}», a * b, a / b, a % b);
        }
```

Результат работы программы:

```
\
1500    6    10
49.7    4.05714285714286    0.1999999999999999
```

6. Сложение (+) и вычитание (-).

Операции сложения и вычитания применимы для целочисленных и вещественных типов данных. Для других типов эти операции применимы, если для них возможно неявное преобразование к целым или вещественным типам.

7. Операции отношения (<, <=, >, >=, ==, !=).
   Операции отношения сравнивают значения левого и правого операндов. Результат операции логического типа: true – если значения совпадают, false – в противном случае. Рассмотрим операции на примере:

```
static void Main()                                  Результат работы
{                                                 программы:
 int i = 15, j = 15;
 Console.WriteLine(i<j); //меньше                    False
 Console.WriteLine(i<=j);  //меньше или равно        True
 Console.WriteLine(i>j);  //больше                   False
 Console.WriteLine(i>=j); //больше или равно         True
 Console.WriteLine(i==j);  //равно                   True
 Console.WriteLine(i!=j);  //не равно                False
}
```

8. Логические операции: И (&&), ИЛИ (||).
   Логические операции применяются к операндам логического типа.

Результат логической операции И имеет значение истина

тогда и только тогда, когда оба операнда принимают значение истина.

Результат логической операции ИЛИ имеет значение истина тогда и только тогда, когда хотя бы один из операндов принимает значение истина.

Рассмотрим операции на примере:

```
static void Main()
        {
          Console.WriteLine(«x     y     x и y    x или y»);
          Console.WriteLine(«{0}  {1}  {2}  {3}», false, false, false&&false, false||false);
          Console.WriteLine(«{0}  {1}  {2}  {3}», false, true, false&&true, false||true);
          Console.WriteLine(«{0}  {1}  {2}  {3}», true, false, true&&false, true||false);
          Console.WriteLine(«{0}  {1}  {2}  {3}», true, true, true&&true, true||true);
         }
```

Результат работы программы:

x y	x и y	x или y	
False	False	False	False
False	True	False	True
True	False	False	True
True	True	True	True

Замечание. Фактически была построена таблица истинности для логических операций И и ИЛИ.

9. Условная операция.

Формат: (<операнд1>)? <операнд2> : <операнд3>;

Операнд1 – это логическое выражение, которое оценивается с точки зрения его эквивалентности константам true и false. Если результат вычисления операнда1 равен true, то результатом условной операции будет значение операнда2, иначе — операнда3. Фактически условная операция является сокращенной формой условного оператора if, который будет рассмотрен позже.

Пример использования условной операции:

```
static void Main()
        {
int x=5; int y=10;
int max = (x > y) ? x : y;
Console.WriteLine(max);
        }
```

10. Операции присваивания: =, +=, -= и т.д.

Формат операции простого присваивания (=):

операнд_2 = операнд_1;

В результате выполнения этой операции вычисляется значение операнда_1, и результат записывается в операнд_2. Возможно связать воедино сразу несколько операторов присваивания, записывая такие цепочки: a=b=c=100. Выражение такого вида выполняется справа налево: результатом выполнения c=100 является число 100, которое затем присваивается переменной b, результатом чего опять является 100, которое присваивается переменной a.

Кроме простой операции присваивания существуют сложные операции присваивания, например, умножение с присваиванием (*=), деление с присваиванием (/=), остаток от деления с присваиванием (%=), сложение с присваиванием (+=), вычитание с присваиванием (-=) и т.д.

В сложных операциях присваивания, например, при сложении с присваиванием, к операнду_2 прибавляется операнд_1, и результат записывается в операнд_2. То есть, выражение c += a является более компактной записью выражения c = c + a . Кроме того, сложные операции присваивания позволяют сгенерировать более эффективный код, за счет того, что в простой операции присваивания для хранения значения правого операнда создается временная

переменная, а в сложных операциях присваивания значение правого операнда сразу записывается в левый операнд.

Рассмотренные операции приведены с учетом убывания приоритета. Если в одном выражении соседствуют операции одного приоритета, то операции присваивания и условная операции выполняются справа налево, а остальные наоборот. Если необходимо изменить порядок выполнения операций, то в выражении необходимо поставить круглые скобки.

Задания к части №2

Задание 2.1. Выясните, допустимы ли следующие способы записи ++(++i), (i--)--, ++(i--) и т.д. И почему.

Задание 2.2. Выясните, допустимы ли следующие способы записи !(-i), -(!a). И почему.

Задание 2.3. Объясните, почему операция (byte)i вместо ожидаемого значения -4 дала нам в качестве результата значение 252.

Задание 2.4. Выясните, чему будет равен результат операции:

1) 1.0/0; 2) 1/0

И объясните, как получился данный результат.

Задание 2.5. Выясните, чему равен результат данного выражения:

1) i<j<k 2) true<false

И объясните, как получился данный результат.

Задание 2.6. Объясните, какое значение примет переменная t в данном фрагменте программы:

```
int a=10, b=3;
bool t=(a>=b && a!=2*b || a<0);
```

Задание 2.7. Измените программу так, чтобы:

```
static void Main()
        {
   int x=5; int y=10;
   int max = (x > y) ? x : y;
   Console.WriteLine(max);
        }
```

1) вычислялось наименьшее значение из двух вещественных чисел x и y;
2) если число двузначное, то на экран выводилось «Да», и «Нет» в противном случае.

Задание 2.8. Объясните, какие значения примут переменные t и b после выполнения данного фрагмента программы:

```
int a=10, b=3;
int t=(a++)-b;
int b+=t*a;
```

Часть 3. Операторы языка C#

Программа на языке C# состоит из последовательности операторов, каждый из которых определяет законченное описание некоторого действия и заканчивается точкой с запятой. Все операторы можно разделить на 4 группы: операторы следования, операторы ветвления, операторы цикла и операторы передачи управления.

1. Операторы следования

Операторы следования выполняются компилятором в естественном порядке: начиная с первого до последнего. К операторам следования относятся: выражение и составной оператор.

Любое выражение, завершающееся точкой с запятой, рассматривается как оператор, выполнение которого заключается в вычислении значения выражения или выполнении законченного действия, например, вызова метода. Например:

```
        ++i;                    //оператор инкремента
        x+=y;                   //оператор сложение с присваива-
нием
        Console.WriteLine(x);   //вызов метода
        x=Math.Pow(a,b)+a*b;    //вычисление сложного выражения
```

Частным случаем оператора выражения является пустой оператор ; Он используется тогда, когда по синтаксису оператор требуется, а по смыслу — нет. В этом случае лишний символ ; является пустым оператором и вполне допустим, хотя и не всегда безопасен. Например, случайный символ ; после условия оператора while или if может совершенно поменять работу этого оператора.

2. Составной оператор или блок представляет собой последовательность операторов, заключенных в фигурные скобки {}.

Блок обладает собственной областью видимости: объявленные внутри блока имена доступны только внутри данного блока или блоков, вложенных в него. Составные операторы применяются в случае, когда правила языка предусматривают наличие только одного оператора, а логика программы требует нескольких операторов. Например, тело цикла while должно состоять только из одного оператора. Если заключить несколько операторов в фигурные скобки, то получится блок, который будет рассматриваться компилятором как единый оператор.

3. Операторы ветвления

Операторы ветвления позволяют изменить порядок выполнения операторов в программе. К операторам ветвления относятся условный оператор if и оператор выбора switch.

3.1.Условный оператор if

Условный оператор if используется для разветвления процесса обработки данных на два направления. Он может

иметь одну из форм: сокращенную или полную.

Форма сокращенного оператора if:

if (B) S;

где B – логическое или арифметическое выражение, истинность которого проверяется; S – оператор: простой или составной.

При выполнении сокращенной формы оператора if сначала вычисляется выражение B, затем проводится анализ его результата: если B истинно, то выполняется оператор S; если B ложно, то оператор S пропускается. Таким образом, с помощью сокращенной формы оператора if можно либо выполнить оператор S, либо пропустить его.

Форма полного оператора if:

if (B) S1; else S2;

где B – логическое или арифметическое выражение, истинность которого проверяется; S1, S2- оператор: простой или составной.

При выполнении полной формы оператора if сначала вычисляется выражение B, затем анализируется его результат: если B истинно, то выполняется оператор S1, а оператор S2 пропускается; если B ложно, то выполняется оператор S2, а S1 – пропускается. Таким образом, с помощью полной формы оператора if можно выбрать одно из двух альтернативных действий процесса обработки данных.

Рассмотрим несколько примеров записи условного оператора if:

```
    if (a > 0) x=y;              // Сокращенная форма с простым оператором
    if (++i) {x=y; y=2*z;}       // Сокращенная форма с составным оператором
```

if (a > 0 || b<0) x=y; else x=z; // Полная форма с простым оператором

if (i+j-1) { x= 0; y= 1;} else {x=1; y:=0;} // Полная форма с составными операторами

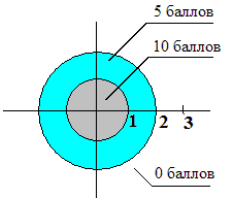
Рассмотрим пример использования условного оператора.

```
static void Main()
        {
    Console.Write(«x= «);
    float x = float.Parse(Console.ReadLine());
    Console.Write(«y=»);
    float y = float.Parse(Console.ReadLine());
    if (x < y ) Console.WriteLine(«min= «+x);
    else Console.WriteLine(«min= «+y);
        }
```

Результат работы программы:

x	y	min
0	0	0
1	-1	-1
-2	2	-2

Операторы S1 и S2 могут также являться операторами if. Такие операторы называют вложенными. При этом ключевое слово else связывается с ближайшим предыдущим словом if, которое еще не связано ни с одним else. Рассмотрим пример программы, использующей вложенные условные операторы.



Пример: Дана мишень.

Подсчитать количество очков после выстрела по данной мишени.

```
static void Main()
            {
    int Ball=0;
    Console.Write(«x= «);
    float x = float.Parse(Console.ReadLine());
Console.Write(«y= «);
    float y = float.Parse(Console.ReadLine());
                if (x * x + y * y <=1) Ball = 10; //окружность с радиусом 1
                 else  if (x * x + y * y <= 4) Ball = 5;     //окружность с радиусом 2
    Console.WriteLine(«Ball= «+ Ball);
            }
```

Результат работы программы:

x	y	Ball
0	0	10
1	-1	5
-2	2	0

3.2.Оператор выбора switch

Оператор выбора switch предназначен для разветвления процесса вычислений по нескольким направлениям. Формат оператора:

```
switch ( <выражение> )
        {
case <константное_выражение_1>:
[<оператор 1>];  <оператор перехода>;
case <константное_выражение_2>:
    [<оператор 2>]; <оператор перехода>;
          ...
        case <константное_выражение_n>:
```

[<оператор n>]; <оператор перехода>;
[default: <оператор>;]
}

Замечание. Выражение, записанное в квадратных скобках, является необязательным элементом в операторе switch. Если оно отсутствует, то может отсутствовать и оператор перехода.

Выражение, стоящее за ключевым словом switch, должно иметь арифметический, символьный, строковый тип или тип указатель. Все константные выражения должны иметь разные значения, но их тип должен совпадать с типом выражения, стоящим после switch или приводиться к нему. Ключевое слово case и расположенное после него константное выражение называют также меткой case.

Выполнение оператора начинается с вычисления выражения, расположенного за ключевым словом switch. Полученный результат сравнивается с меткой case. Если результат выражения соответствует метке case, то выполняется оператор, стоящий после этой метки, за которым обязательно должен следовать оператор перехода: break, goto и т.д. При использовании оператора break происходит выход из switch и управление передается оператору, следующему за switch. Если же используется оператор goto, то управление передается оператору, помеченному меткой, стоящей после goto.

Пример. По заданному виду арифметической операции (сложение, вычитание, умножение и деление) и двум операндам, вывести на экран результат применения данной операции к операндам.

```
static void Main()
        {
Console.Write(«OPER= «);
char oper=char.Parse(Console.ReadLine());
```

```
bool ok=true;
Console.Write(«A= «);
int a=int.Parse(Console.ReadLine());
Console.Write(«B= «);
int b=int.Parse(Console.ReadLine());
float res=0;
switch (oper)
            {
case '+': res = a + b; break;          //1
case '-': res = a - b; break;
case '*': res = a * b; break;
case ':': if (b != 0)
                        {
res = (float)a / b; break;
                        }
else goto default;
default: ok = false; break;
            }
if (ok) Console.WriteLine(«{0} {1} {2} = {3}», a, oper, b,
res);
else Console.WriteLine(«error»);
        }
```

Результат выполнения программы:

oper	x	y	rez
• 4	5	9	
:	4	0	error
%	4	3	error

Если необходимо, чтобы для разных меток выполнялось одно и тоже действие, то метки перечисляются через

двоеточие. Например:

```
switch (oper)
                {
    case '+': res = a + b; break;
    case '-': res = a - b; break;
    case '*': res = a * b; break;
    case '/': case ':': if (b != 0)  // перечисление меток
                                    {
    res = (float)a / b; break;
                                    }
    else goto default;
    default: ok = false; break;
        }
```

4. Операторы цикла

Операторы цикла используются для организации многократно повторяющихся вычислений. К операторам цикла относятся: цикл с предусловием while, цикл с постусловием do while, цикл с параметром for и цикл перебора foreach..

4.1.Цикл с предусловием while

Оператор цикла while организует выполнение одного оператора (простого или составного) неизвестное заранее число раз. Формат цикла while:

while (B) S;

где B – выражение, истинность которого проверяется (условие завершения цикла); S – тело цикла - оператор (простой или составной).

Перед каждым выполнением тела цикла анализируется значение выражения B: если оно истинно, то выполняется

тело цикла, и управление передается на повторную проверку условия B; если значение B ложно – цикл завершается и управление передается на оператор, следующий за оператором S.

Если результат выражения B окажется ложным при первой проверке, то тело цикла не выполнится ни разу. Отметим, что если условие B во время работы цикла не будет изменяться, то возможна ситуация зацикливания, то есть невозможность выхода из цикла. Поэтому внутри тела должны находиться операторы, приводящие к изменению значения выражения B так, чтобы цикл мог корректно завершиться.

В качестве иллюстрации выполнения цикла while рассмотрим программу вывода на экран целых чисел из интервала от 1 до n.

```
static void Main()
        {
Console.Write(«N= «);
int n=int.Parse(Console.ReadLine());
int i = 1;
while (i <= n)          //пока i меньше или равно n
    Console.Write(« «+ i++ ); //выводим i на экран, затем
    увеличиваем его на 1
        }
```

Результаты работы программы: n ответ
10 1 2 3 4 5 6 7 8 9 10

4.2.Цикл с постусловием do while

Оператор цикла do while также организует выполнение одного оператора (простого или составного) неизвестное заранее число раз. Однако в отличие от цикла while условие завершения цикла проверяется после выполнения тела

цикла. Формат цикла do while:

do S while (B);

где B – выражение, истинность которого проверяется (условие завершения цикла); S – тело цикла - оператор (простой или блок).

Сначала выполняется оператор S, а затем анализируется значение выражения B: если оно истинно, то управление передается оператору S, если ложно - цикл завершается, и управление передается на оператор, следующий за условием B. Так как условие B проверяется после выполнения тела цикла, то в любом случае тело цикла выполнится хотя бы один раз.

В операторе do while, так же как и в операторе while, возможна ситуация зацикливания в случае, если условие B всегда будет оставаться истинным.

В качестве иллюстрации выполнения цикла do while рассмотрим программу вывода на экран целых чисел из интервала от 1 до n.

```
static void Main()
        {
    Console.Write(«N= «);
    int n=int.Parse(Console.ReadLine());
    int i = 1;
    do
    Console.Write(« « + i++); //выводим i на экран, затем
        увеличиваем его на 1
    while (i <= n);          //пока i меньше или равно n
        }
```

4.3.Цикл с параметром for

Цикл с параметром имеет следующую структуру:

for (<инициализация>; <выражение>; <модификация>)

<оператор>;

Инициализация используется для объявления и/или присвоения начальных значений величинам, используемым в цикле в качестве параметров (счетчиков). В этой части можно записать несколько операторов, разделенных запятой. Областью действия переменных, объявленных в части инициализации цикла, является цикл и вложенные блоки. Инициализация выполняется один раз в начале исполнения цикла.

Выражение определяет условие выполнения цикла: если его результат истинен, цикл выполняется. Истинность выражения проверяется перед каждым выполнением тела цикла, таким образом, цикл с параметром реализован как цикл с предусловием. В блоке выражение через запятую можно записать несколько логических выражений, тогда запятая равносильна операции логическое И (&&).

Модификация выполняется после каждой итерации цикла и служит обычно для изменения параметров цикла. В части модификация можно записать несколько операторов через запятую.

Оператор (простой или составной) представляет собой тело цикла.

Любая из частей оператора for (инициализация, выражение, модификация, оператор) может отсутствовать, но точку с запятой, определяющую позицию пропускаемой части, надо оставить.

```
static void Main()
        {
    Console.Write(«N= «);
    int n=int.Parse(Console.ReadLine());
    for (int i=1; i<=n;)     //блок модификации пустой
    Console.Write(« « + i++);
        }
```

Замечание. Цикл перебора foreach будет рассмотрен позже.

Вложенные циклы

Циклы могут быть простые или вложенные (кратные, циклы в цикле). Вложенными могут быть циклы любых типов: while, do while, for. Каждый внутренний цикл должен быть полностью вложен во все внешние циклы. «Пересечения» циклов не допускаются.

Рассмотрим пример использования вложенных циклов, который позволит вывести на экран числа следующим образом:

```
static void Main()

        {
for (int i = 1; i <= 4; ++i, Console.WriteLine())     //1

    for (int j=1; j<=5; ++j)
    Console.Write(« « + 2);
        }
```

Замечание. В строке 1 в блоке модификации содержится два оператора ++i и Console.WriteLine(). В данном случае после каждого увеличения параметра i на 1 курсор будет переводиться на новую строку.

5. Операторы безусловного перехода
   В C# есть несколько операторов, изменяющих естественный порядок выполнения команд: оператор безусловного перехода goto, оператор выхода break, оператор перехода к следующей итерации цикла continue, оператор возврата из метода return и оператор генерации исключения throw.

5.1.Оператор безусловного перехода goto

Оператор безусловного перехода goto имеет формат:

goto <метка>;

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

<метка>: <оператор>;

Оператор goto передает управление на помеченный меткой оператор. Рассмотрим пример использования оператора goto:

```
static void Main()

            {
            float x;
            metka: Console.WriteLine(«x=»);     //оператор, помеченный меткой
               x = float.Parse(Console.ReadLine());
               if (x!=0) Console.WriteLine(«y({0})={1}», x, 1 / x );
               else
                     {
      Console.WriteLine(«функция не определена»);
      goto metka;// передача управление метке
                     }
             }
```

Следует учитывать, что использование оператора goto затрудняет чтение больших по объему программ, поэтому использовать метки нужно только в крайних случаях, например, в операторе switch.

5.2.Оператор выхода break

Оператор break используется внутри операторов ветвления и цикла для обеспечения перехода в точку программы, находящуюся непосредственно за оператором, внутри которого находится break.

Мы уже применяли оператор break для выхода из оператора switch, аналогичным образом он может применяться для выхода из других операторов.

5.3.Оператор перехода к следующей итерации цикла continue

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

(повторение тела цикла). Рассмотрим оператор continue на примере.

```
static void Main()

        {
    Console.WriteLine(«n=»);
    int n = int.Parse(Console.ReadLine());
    for (int i = 1; i <= n; i++)
        {
    if (i % 2 == 0) continue;
    Console.Write(« « + i);
        }
    }
```

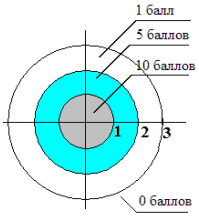
Замечание. Операторы return и throw будут рассмотрены позже.

Задания к части №3

Задание 3.1. Измените программу так, чтобы вычислялось наибольшее значение из x и y.

```
static void Main()

        {
    Console.Write("x= ");
    float x = float.Parse(Console.ReadLine());
    Console.Write("y=");
    float y = float.Parse(Console.ReadLine());
    if (x < y ) Console.WriteLine("min= "+x);
    else Console.WriteLine("min= "+y);
        }
```

Задание 3.2. Измените программу так, чтобы подсчитывалось количество очков для мишени вида




```
static void Main()

        {
    int Ball=0;
    Console.Write("x= ");
    float x = float.Parse(Console.ReadLine());
    Console.Write("y= ");
```

```
float y = float.Parse(Console.ReadLine());
                if (x * x + y * y <=1) Ball = 10;          //окружность с радиусом 1
                 else  if (x * x + y * y <= 4) Ball = 5;   //окружность с радиусом 2
Console.WriteLine("Ball= "+ Ball);

            }
```

Задания 3.3.

1. Замените в строке 1 оператор break, на оператор goto case '-' и посмотрите, что произойдет, если в качестве операции ввести +.
2. В условном операторе if уберите ветку else и посмотрите, что произойдет.

```
static void Main()

            {
Console.Write("OPER= ");
char oper=char.Parse(Console.ReadLine());
bool ok=true;
Console.Write("A= ");
int a=int.Parse(Console.ReadLine());
Console.Write("B= ");
int b=int.Parse(Console.ReadLine());
float res=0;
switch (oper)
                {
case '+': res = a + b; break;        //1
case '-': res = a - b; break;
case '*': res = a * b; break;
case ':': if (b != 0)
                                {
res = (float)a / b; break;

                                }
else goto default;

default: ok = false; break;

                }
if (ok) Console.WriteLine("{0} {1} {2} = {3}", a, oper, b, res);

else Console.WriteLine("error");

            }
```

Задание 3.4. Измените программу так, чтобы:

1. числа выводились в обратном порядке;
2. выводились только нечетные числа.

```
static void Main()

            {
Console.Write("N= ");
int n=int.Parse(Console.ReadLine());
int i = 1;
while (i <= n)          //пока i меньше или равно n
```

```
Console.Write(« «+ i++ ); //выводим i на экран, затем увеличиваем его на 1
        }
```

Задание 3.5. Измените программу так, чтобы:

1. числа выводились в обратном порядке;
2. выводились только четные числа.

```
static void Main()
        {
Console.Write("N= ");
int n=int.Parse(Console.ReadLine());
int i = 1;
do
Console.Write(« « + i++); //выводим i на экран, затем увеличиваем его на 1
while (i <= n);          //пока i меньше или равно n
        }
```

Задание 3.6. Измените программу так, чтобы:

1. числа выводились в обратном порядке;
2. выводились квадраты чисел.

```
static void Main()
        {
Console.Write("N= ");
int n=int.Parse(Console.ReadLine());
for (int i=1; i<=n;)     //блок модификации пустой
Console.Write(" " + i++);
        }
```

Задание 3.7. Измените программу так, чтобы таблица содержала n и m столбцов (значения n и m вводятся с клавиатуры).

```
static void Main()
        {
for (int i = 1; i <= 4; ++i, Console.WriteLine())     //1
    for (int j=1; j<=5; ++j)
    Console.Write(" " + 2);
        }
```

Замечание. В строке 1 в блоке модификации содержится два оператора ++i и Console.WriteLine(). В данном случае после каждого увеличения параметра i на 1 курсор будет переводиться на новую строку.

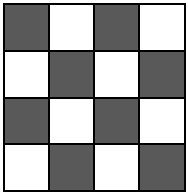
Секция самостоятельной (домашней) работы

Домашнее задание №1.

Рассмотреть самостоятельно сдвиг влево (<<), сдвиг вправо (>>); поразрядные операции И (&), исключающее ИЛИ (^) и ИЛИ (|); сложные операции присваивания: <<=, >>=, &=, ^=, |=.

Домашнее задание №2

Задача 2.1. Дана шахматная доска размером n□n клеток. Верхняя левая клетка доски черная и имеет номер (1, 1). Например, для n=4 шахматная таблица выглядит следующим образом:



1. для заданного значения n определить количество черных ячеек шахматной доски;
2. по номеру ячейки (k, m) определить ее цвет;
3. определить, являются ли ячейки с номерами (k1, m1) и (k2, m2) одного цвета;
4. определить, находится ли фигура, стоящая в ячейке с номером (k1, m1), под ударом второй фигуры, стоящей в ячейке с номером (k2, m2), при условии, что ход второй фигуры и ей является: a) пешка; b) слон; c) ладья; d) ферзь; e) конь.

Задача 2.2. Задана дата в формате <день>.<месяц>.<год>. Определить:

1) сколько дней прошло с начала года; 2) сколько дней осталось до конца года;

3) дату предыдущего дня; 4) дату сле-

дующего дня.

Задача 2.3. Натуральное число из n цифр является числом Армстронга, если сумма его цифр, возведенных в n-ную степень, равна самому числу. Например, 153=13+53+33. Найти все трехзначные числа Армстронга.

Задача 2.4. Стороны прямоугольника заданы натуральными числами n и m. Найти количество квадратов (стороны которых выражены натуральными числами), на которые можно разрезать данный прямоугольник, если от него каждый раз отрезать квадрат:

1) наименьшей площади; 2) наибольшей площади

ЛАБОРАТОРНАЯ РАБОТА №3. РАБОТА С МАССИВАМИ В C#

Задание 1. Объявление и инициализация одномерных массивов.

1. Создайте новое консольное приложение `ArrayTest`;
2. Добавьте в проект класс Arrs, для реализации базовых функций работы с массивами. В частности в созданном классе определите:
   a. Статическое закрытое поле rnd типа Random и инициализируйте его, вызвав соответствующий конструктор;
   b. Создайте открытый статический метод CreateOneDimAr, получающий в качестве параметра массив целых чисел и инициализирующий его случайными значениями

      Методические указания. Для инициализации можно воспользоваться методом Next(int,int) объекта rnd. Для определения количества элементов в массиве воспользуйтесь методом GetLength());
   c. Создайте открытый статический метод PrintArr1, получающий два параметра: (string – название переданного массива, int[] – массив) и выводящий

значения элементов массива на консоль.

3. В теле функции main выполните следующие действия:
   a. Объявите три одномерных массива A, B, C состоящих из 5-ти элементов типа int:
      Методические указания. `int[] A = new int[5], B = new int[5], C = new int[5];`
   b. Для инициализации массивов A и B вызовите метод CreateOneDimAr;
   c. Массив C инициализируйте значениями суммы соответствующих элементов массивов A и B;
   d. Объявите массив X с явной инициализацией: `int[] X = { 5,5,6,6,7,7};`
   e. Объявите два массива U и V с отложенной инициализацией: `int[] U, V;`
   f. Выделите память для массива U для хранения трёх элементов и инициализируйте его значениями от 1 до 3;
   g. Запишите присваивание массиву V присваивание константного массива {1,2,3}. Как Вы думаете, почему данное присвоение недопустимо? Закомментируйте строку с недопустимым присвоением.
   h. Выделите память для массива V для хранения трёх элементов и присвойте значение массива U: `V = U;`
   i. Измените значение первого элемента массива V на 9;
   j. С помощью функции `PrintArr1` Выведите массивы A, B, C, X, U, V.
4. Запустите полученную программу. Прокомментируйте полученные результаты.

Во всех вышеприведенных примерах объявлялись статические массивы, поскольку нижняя граница равна нулю по определению, а верхняя всегда задавалась в этих примерах константой. Напомню, что в C# все массивы, независимо от того, каким выражением описывается граница, рассматриваются как динамические, и память для них распределяется в "куче". Полагаю, что это отражение разумной точки зрения: ведь статические массивы, скорее исключение, а правилом является использование динамических массивов. В действительности реальные потребности в

размере массива, скорее всего, выясняются в процессе работы в диалоге с пользователем.

Чисто синтаксически нет существенной разницы в объявлении статических и динамических массивов. Выражение, задающее границу изменения индексов, в динамическом случае содержит переменные. Единственное требование - значения переменных должны быть определены в момент объявления. Это ограничение в C# выполняется автоматически, поскольку хорошо известно, сколь требовательно C# контролирует инициализацию переменных.

Рассмотрим пример, в котором описана работа с динамическим массивом:

5. Запросите у пользователя размерность динамического массива:

   Методические указания:

```
Console.WriteLine("Введите размерность массива:");
int size = int.Parse(Console.ReadLine());
```

6. Опишите массив D и укажите для него размерность введенную пользователем, инициализируйте массив с помощью функции CreateOneDimAr, и выведите его значения с помощью функции PrintArr1;

Задание 2. Многомерные массивы

Разделение массивов на одномерные и многомерные носит исторический характер. Никакой принципиальной разницы между ними нет. Одномерные массивы - это частный случай многомерных. Можно говорить и по-другому: многомерные массивы являются естественным обобщением одномерных.

Одномерные массивы позволяют задавать такие математические структуры как векторы, двумерные - матрицы, трехмерные - кубы данных, массивы большей размерности - многомерные кубы данных.

В чем особенность объявления многомерного массива? Как в типе указать размерность массива? Это делается достаточно просто, за счет использования запятых. Вот как выглядит объявление многомерного массива в общем случае:

<тип>[, ... ,] <объявители>;

Число запятых, увеличенное на единицу, и задает размерность массива. Что касается объявителей, то все, что сказано для одномерных массивов, справедливо и для многомерных. Можно лишь отметить, что хотя явная инициализация с использованием многомерных константных массивов возможна, но применяется редко из-за громоздкости такой структуры. Проще инициализацию реализовать программно, но иногда она все же применяется.

Пример:

```
int[,]matrix = {{1,2},{3,4}};
```

1. Написать функцию MultMatr умножения матриц:
    a. Функция получает две матрицы, произвольного размера;
    b. Функция возвращает результирующую матрицу в случае, если умножение возможно, и пустую матрицу, если не возможно;
    c. Для определения размерности матрицы можно воспользоваться методом GetLength(), у которой в качестве параметра указывается номер измерения (счет измерений начинается с нуля);
2. В классе Arrs напишите функцию PrintArr2 для вывода матрицы на печать;
3. В классе Arrs напишите функцию CreateAr2 заполнения двумерного массива произвольными числами (по аналогии с одномерным массивом);
4. Создайте две матрицы размерностью 3*3. Инициализируйте их. Выполните функцию умножения и значения всех трёх матриц выведите на экран.

Задание 3. Массивы массивов.

Еще одним видом массивов C# являются массивы массивов, называемые также **изрезанными массивами (jagged arrays).** Такой массив массивов можно рассматривать как одномерный массив, элементы которого являются массивами, элементы которых, в свою очередь, снова могут быть массивами, и так может продолжаться до некоторого уровня вложенности.

В каких ситуациях может возникать необходимость в таких структурах данных? Эти массивы могут применяться для представления деревьев, у которых узлы могут иметь произвольное число потомков. Таковым может быть, например, генеалогическое

дерево. Есть некоторые особенности в объявлении и инициализации таких массивов. Если при объявлении типа многомерных массивов для указания размерности использовались запятые, то для изрезанных массивов применяется более ясная символика - совокупности пар квадратных скобок; например, int[][] задает массив, элементы которого - одномерные массивы элементов типа int.

Сложнее с созданием самих массивов и их инициализацией. Здесь нельзя вызвать **конструктор new int[3][5],** поскольку он не задает изрезанный массив. **Фактически нужно вызывать конструктор для каждого массива на самом нижнем уровне.** В этом и состоит сложность объявления таких массивов. Рассмотрим формальный пример:

```
//массив массивов - формальный пример
//объявление и инициализация
int[][] jagger = new int[3][]
{
   new int[] {5,7,9,11},
   new int[] {2,8},
   new int[] {6,12,4}
};
```

Массив jagger имеет всего два уровня. Можно считать, что у него три элемента, каждый из которых является массивом. Для каждого такого массива необходимо вызвать конструктор new, чтобы создать внутренний массив. В данном примере элементы внутренних массивов получают значение, будучи явно инициализированы константными массивами. Конечно, допустимо и такое объявление:

```
int[][] jagger1 = new int[3][]
{
   new int[4],
   new int[2],
   new int[3]
};
```

В этом случае элементы массива получат при инициализации нулевые значения. Реальную инициализацию нужно будет выполнять программным путем. Стоит заметить, что в конструкторе верхнего уровня константу 3 можно опустить и писать про-

сто new int[][]. Самое забавное, что вызов этого конструктора можно вообще опустить - он будет подразумеваться:

```
int[][] jagger2 =
{
   new int[4],
   new int[2],
   new int[3]
};
```

А вот конструкторы нижнего уровня необходимы. Еще одно важное замечание - динамические массивы возможны и здесь. В общем случае, границы на любом уровне могут быть выражениями, зависящими от переменных. Более того, допустимо, чтобы массивы на нижнем уровне были многомерными. Но это уже "от лукавого" - вряд ли стоит пользоваться такими сложными структурами данных, ведь с ними предстоит еще и работать.

1. В классе Arrs напишите функцию PrintArr3 для вывода двумерного массива массивов;

   Методические указания: Поскольку в данном случае используется не «матрица», а массив массивов, то логично вместо функции GetLength() использовать свойство Length для соответствующего массива.
2. В классе Arrs напишите функцию CreateAr3 заполнения двумерного массива произвольными числами (по аналогии с одномерным массивом);
3. Опишите массив R размерностью 10 массивов, каждый из которых на 1 больше предыдущего; (R[0] – содержит массив из одного элемента, а R[5] – массив из шести элементов)
4. Инициализируйте массив R случайными значениями;
5. Выведите его содержимое на консоль;

Контрольные вопросы

1. Почему для работы с массивами в данной лабораторной работе, в основном, используются статические методы?
2. Как можно определить размер переданного в функцию массива?
3. Можно ли в массивах на языке C# использовать индексацию начинающуюся с 1?
4. Можно ли создать массив без его инициализации?

5. Разрешена ли операция присваивания между массивами?
6. Как в языке C# объявить динамический массив? Как объявить статический?
7. Как описывается многомерный массив? Как определяется его размерность?
8. Чем отличаются многомерные массивы от изрезанных массивов?
9. Каковы отличия свойства Length и метода GetLength с точки зрения возвращаемого значения?

ЛАБОРАТОРНАЯ РАБОТА №4. РАБОТА С МАССИВАМИ КАК С КОЛЛЕКЦИЯМИ В C#

Задание 1. Класс Array

Нельзя понять многие детали работы с массивами в C#, если не знать устройство класса Array из библиотеки FCL, потомками которого являются все классы-массивы.

Рассмотрим следующие объявления:

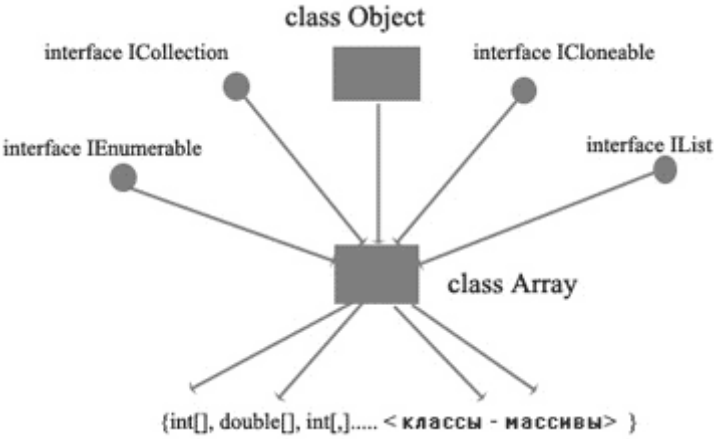
```
//Класс Array
        int[] ar1 = new int[5];
        double[] ar2 ={5.5, 6.6, 7.7};
        int[,] ar3 = new Int32[3,4];
```

Зададимся естественным вопросом: к какому или к каким классам принадлежат объекты ar1, ar2 и ar3? Ответ прост: все они принадлежат к разным классам. Переменная ar1 принадлежит к классу int[] - одномерному массиву значений типа int, ar2 - double[] - одномерному массиву значений типа double, ar3 - двумерному массиву значений типа int.

Следующий закономерный вопрос: а что общего есть у этих трех объектов? Прежде всего, все три класса этих объектов, как и другие классы, являются потомками класса Object, а потому имеют общие методы, наследованные от класса Object и доступные объектам этих классов.

У всех классов, являющихся массивами, много общего, поскольку все они являются потомками класса System.Array. Класс System.Array наследует ряд интерфейсов: ICloneable, IList, ICollection, IEnumerable, а, следовательно, обязан реализовать все

их методы и свойства. Помимо наследования свойств и методов класса Object и вышеперечисленных интерфейсов, класс Array имеет довольно большое число собственных методов и свойств. Взгляните, как выглядит отношение наследования на семействе классов, определяющих массивы.



Благодаря такому мощному родителю, над массивами определены самые разнообразные операции - копирование, поиск, обращение, сортировка, получение различных характеристик.

Массивы можно рассматривать как коллекции и устраивать циклы foreach для перебора всех элементов. Важно и то, что когда у семейства классов есть общий родитель, то можно иметь общие процедуры обработки различных потомков этого родителя. Для общих процедур работы с массивами характерно, что один или несколько формальных аргументов имеют родительский тип Array. Естественно, внутри такой процедуры может понадобиться анализ - какой реальный тип массива передан в процедуру.

Выполните следующее задание:

1. В проект лабораторной работы по массивам №1 добавьте универсальную функцию PrintAnyArr (печать любого массива, кроме усеченных);

Методические указания: Второй параметр функции должен быть типа Array. Для того, чтобы отличить размерность массива (отличить вектор от матрицы) слудует использовать свойство Rank. К элементам массива A, имеющего

класс Array, нет возможности прямого доступа в обычной манере - A [<индексы>], но зато есть специальные методы GetValue (<индексы>) и SetValue (<индексы>).

2. Дополнительно. Попробуйте модифицировать функцию, таким образом чтобы она могла печатать усеченные массивы. (10 баллов)

Задание 2. Массивы как коллекции. Статические методы класса Array.

В ряде задач массивы C# целесообразно рассматривать как коллекции, не используя систему индексов для поиска элементов. Это, например, задачи, требующие однократного или многократного прохода по всему массиву - нахождение суммы элементов, нахождение максимального элемента, печать элементов. В таких задачах вместо циклов типа For по каждому измерению достаточно рассмотреть единый цикл For Each по всей коллекции. Эта возможность обеспечивается тем, что класс Array наследует интерфейс IEnumerable. Обратите внимание, этот интерфейс обеспечивает только возможность **чтения элементов коллекции (массива), не допуская их изменения.**

1. Примените эту стратегию и постройте еще одну версию процедуры печати PrintAnyArr2. Эта версия должна быть самой короткой и самой универсальной, поскольку подходит для печати массива, независимо от его размерности и типа элементов.

Конечно, за все нужно платить. Платой за универсальность процедуры печати является то, что многомерный массив печатается как одномерный без разделения элементов на строки.

Статические методы класса Array позволяют решать самые разнообразные задачи:

Copy - позволяет копировать весь массив или его часть в другой массив.

IndexOf, LastIndexOf - определяют индексы первого и последнего вхождения образца в массив, возвращая -1, если такового вхождения не обнаружено.

Reverse - выполняет обращение массива, переставляя элементы в обратном порядке.

Sort - осуществляет сортировку массива.

BinarySearch - определяет индекс первого вхождения образца в отсортированный массив, используя алгоритм двоичного поиска.

1. Напишите код метода, реализующий примеры работы всех перечисленных выше методов.

Задание 3. Класс Object и массивы

Подведем некоторые итоги:

Свойства класса Array		
Свойство	**Родитель**	**Описание**
IsFixedSize	Интерфейс IList	True, если массив статический
IsReadOnly	Интерфейс IList	Для всех массивов имеет значение false
IsSynchronized	Интерфейс ICollection	True или False, в зависимости от того, установлена ли синхронизация доступа для массива

SyncRoot	Интерфейс ICollection	Собственный метод синхронизации доступа к массиву. При работе с массивом его можно закрыть на время обработки, что запрещает его модификацию каким-либо потоком:
		`Array myCol = new int[];`
		`lock( myCol.SyncRoot ) {`
		`foreach ( Object item in myCol )`
		`{`
		`// безопасная обработка массива }`

Length		Число элементов массива
Rank		Размерность массива

Статические методы класса Array	
Метод	**Описание**
BinarySearch	Двоичный поиск. Описание и примеры даны в тексте
Clear	Выполняет начальную инициализацию элементов. В зависимости от типа элементов устанавливает значение 0 для арифметического типа, false - для логического типа, Null для ссылок, "" - для строк.
Copy	Копирование части или всего массива в другой массив. Описание и примеры даны в тексте
CreateInstance	Класс Array, в отличие от многих классов, может создавать свои экземпляры не только с помощью конструктора new, но и при вызове метода CreateInstance: `Array my2Dar = Array.CreateInstance(typeof(double), 2,2)`
IndexOf	Индекс первого вхождения образца в массив.
LastIndexOf	Индекс последнего вхождения образца в массив.
Reverse	Обращение одномерного массива.
Sort	Сортировка массива.

Динамические методы класса Array

Метод	**Родитель**	**Описание**
Equals	Класс Object	Описание и примеры даны в предыдущих главах.
GetHashCode	Класс Object	Описание и примеры даны в предыдущих главах.
GetType	Класс Object	Описание и примеры даны в предыдущих главах.
ToString	Класс Object	Описание и примеры даны в предыдущих главах.

Clone	Интерфейс ICloneable	Позволяет создать плоскую или глубокую копию массива. В первом случае создаются только элементы первого уровня, а ссылки указывают на те же самые объекты. Во втором случае копируются объекты на всех уровнях. Для массивов создается только плоская копия.
CopyTo	Интерфейс ICollection	Копируются все элементы одномерного массива в другой одномерный массив, начиная с заданного индекса: `col1.CopyTo(col2,0);`
GetEnumerator	Интерфейс IEnumerable	Стоит за спиной цикла foreach
GetLength		Возвращает число элементов массива по указанному измерению. Описание и примеры даны в тексте главы.
GetLowerBound, GetUpperBound		Возвращает нижнюю и верхнюю границу по указанному измерению. Для массивов нижняя граница всегда равна нулю.
GetValue, SetValue		Возвращает или устанавливает значение элемента массива с указанными индексами.
Initialize		Может быть применен только к массивам значимого типа. Инициализирует элементы, вызывая соответствующий конструктор. Как правило, не используется в обычных программах.

1. Приведите пример показывающий отличие между возвращаемым значением метода GetLength и свойством Length.
2. Напишите функцию печати массива, принимающую в качестве параметра объект типа Object.

Задание 4. Массивы Объектов.

Во всех рассмотренных примерах нам встречались массивы, элементы которых имели только простые значимые типы. В реальных программах массивы объектов и других ссылочных типов встречаются не менее часто. Каков бы ни был тип элементов, большой разницы при работе с массивами нет. Но один важный нюанс все же есть, и его стоит отметить. Он связан с инициализацией элементов по умолчанию. Уже говорилось о том, что компилятор не следит за инициализацией элементов массива и доверяет инициализации, выполненной конструктором массива по умолчанию. Но для массивов ссылочного типа инициализация по умолчанию присваивает **ссылкам значение Null**. Это означает, что создаются только ссылки, но не сами объекты. По этой причине, пока не будет проведена настоящая инициализация с созданием объектов и заданием ссылок на конкретные объекты, работать с массивом ссылочного типа будет невозможно.

ассмотрим детали этой проблемы на примере. Определим достаточно простой и интуитивно понятный класс, названный Student свойства которого задают имя студента и его баллы, а методы позволяют установить количество полученных баллов и распечатать его свойства.

1. Создайте новое консольное приложение;
2. Определите новый класс Student с двумя закрытыми полями полями: name и sumb;
3. Объявите статическое или динамическое поле rnd и инициализируйте его объектом класса Random;
4. Напишите открытый метод SetValue, которому передается фамилия студента и который инициализирует соответствующее поле, а так же с помощью объекта rnd инициализирует значение набранных баллов случайным числом.
5. Напишите функцию вывода информации об объекте на консоль, предварительно переопределив метод ToString;
6. В функции main создайте массив объектов Student из трёх элементов с отложенной инициализацией;
7. Задайте значения полей элементов данного массива , используя специальную функцию SetValue; Когда будет сгенерировано сообщение об ошибке? Почему?

8. Для решения указанной проблемы чаще всего используют следующий приём:
    a. В классе Student определите метод InitAr (Метод должен быть статическим, чтобы его можно было вызывать еще до того, как созданы экземпляры класса, поскольку метод предназначен для создания этих самых экземпляров.) Метод имеет следующую сигнатуру:
        public static Student[] InitAr(Student[] Stud)
    b. В цикле инициализируются все элементы массива с помощью констуктора по умолчанию Student();
    c. Не забудьте вернуть сам массив!
9. Вызовите данный метод в функции main и снова попробуйте выполнить шаг №7;
10. Выведите массив на экран;

Задание 5. Приведение типов массивов.

Преобразования между классами массивов и родительскими классами Array и Object уже рассматривались. А существуют ли другие преобразования между классами массивов? Что происходит при присваивании x=e; (передаче аргументов в процедуру), если x и e - это массивы разных классов? Возможно ли присваивание? Ответ на этот вопрос положительный, хотя накладываются довольно жесткие ограничения на условия, когда такие преобразования допустимы.

Известно, например, что между классами Int и Object существуют взаимные преобразования - в одну сторону явное, в другую неявное. А вот между классами Int[] и Object[] нет ни явных, ни неявных преобразований. С другой стороны, такое преобразование существует между классами String[] и Object[].

В чем же тут дело, и где логика? Запомните, главное ограничение на возможность таких преобразований состоит в том, что **элементы массивов должны иметь ссылочный тип**. А теперь притянем сюда логику. Крайне желательно обеспечить возможность проведения преобразований между массивами, элементы которых принадлежат одному семейству классов, связанных отношением наследования. Такая возможность и была реализована. А вот для массивов с элементами значимых типов подоб-

ную же возможность не захотели или не смогли реализовать.

Сформулируем теперь точные правила, справедливые для присваивания и передачи аргументов в процедуру. Для того, чтобы было возможным неявное преобразование массива с элементами класса S в массив с элементами класса T, необходимо выполнение следующих условий:

- классы S и T должны быть ссылочного типа;
- размерности массивов должны совпадать;
- должно существовать неявное преобразование элементов класса S в элементы класса T.

Заметьте, если S - это родительский класс, а T - его потомок, то для массивов одной размерности остальные условия выполняются. Вернемся теперь к примеру с классами Int[], String[] и Object[]. Класс Int не относится к ссылочным классам, и потому преобразования класса Int[] в Object[] не существует. Класс string является ссылочным классом и потомком класса Object, а потому существует неявное преобразование между классами String[] и Object[].

Правило для явного преобразования можно сформулировать, например, так. Если существует неявное преобразование массива с элементами класса S в массив с элементами класса T, то существует явное преобразование массива с элементами класса T в массив с элементами класса S.

1. Откройте консольное приложение созданное в задании 1;
2. Добавьте в класс для работы с массивами еще одну функцию печати массива объектов со следующей сигнатурой: public static void PrintArObj(string name,object[] A)
3. Реализуйте тело этой функции воспользовавшись циклом foreach;
4. Протестируйте работу данной функции для массивов различных типов;

Задание 6. Описание класса массива

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 вычита-

ния 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.

Контрольные вопросы

1. Какие классы наследуют класс Array? Что дает такое наследование?
2. Перечислите основные интерфейсы реализованные в Array;
3. Можно ли модифицировать значения элемента массива когда он передан как Array&?
4. Можно ли модифицировать значения элемента массива, когда работа с ним осуществляется как с коллекцией элементов посредством цикла foreach?
5. Перечислите параметры метода SetValue класса Array;
6. Можно ли массивы приводить к классу Obect? А обратно?
7. На что в задании 4 повлияет выбор между статическим и динамическим полем rnd?
8. Когда происходит инициализация объектов массива. Какие пути решения проблемы инициализации можете предложить?
9. Сформулируйте правила преобразования ссылок на массивы.
10. Укажите возможны ли следующие преобразования:
    a. int[] в object[];
    b. string[] в object[];
    c. object[] в sting[];

ЛАБОРАТОРНАЯ РАБОТА №5. РАБОТА СО СТРОКАМИ В C#.

С точки зрения ежедневного программирования одним

из самых важных типов данных C# является тип string. Он предназначен для определения и поддержки символьных строк. Во многих других языках программирования строка представляет собой массив символов. В C# дело обстоит иначе: здесь строки являются объектами. Таким образом, string – это ссылочный тип. Несмотря на то что string – встроенный тип данных, для его рассмотрения необходимо иметь представление о классах и объектах.

Задание 1.

Объявление и инициализация строк. Использование простейших операций со строками.

Класс string содержит ряд методов, которые предназначены для обработки. Тип string также включает свойство Length, которое содержит длину строки. Чтобы получить значение отдельного символа строки, достаточно использовать индекс. Например:

```
string str = " t e s t " ;
Console.WriteLine(string[0]);
```

При выполнении этого фрагмента программы на экран будет выведен символ t (первый символ слова "test"). Как и у массивов, индексация строк начинается с нуля. Однако здесь важно отметить, что с помощью индекса нельзя присвоить значение символу внутри строки (**Объяснить почему???**).

Чтобы узнать, равны ли две строки, необходимо использовать оператор "==". Обычно, когда оператор "==" применяется к ссылочным объектам, он определяет, относятся ли обе ссылки к одному и тому же объекту. Но применительно к объектам типа string дело обстоит иначе. В этом случае проверяется равенство содержимого двух строк (**Объяснить почему???**). То же справедливо и в отноше-

нии оператора " ! = " . Что касается остальных операторов отношения (например, ">" или ">="), то они сравнивают ссылки так же, как и объекты других типов.

1. Создайте консольное приложение StringExample.
2. Создайте три объекта типа string, использовав при этом три различных вида инициализации.
   Например:
   - string str1 = ”Строка1”;
   - string str2 = new string (‘s’, 5); //объяснить синтаксис и назначение этого конструктора
   - char[] charray={‘t’,’e’,’s’,’t’};
     string str3 =new string (charray);
3. Протестируйте на созданных строках простейшие операции:
   - присваивание (=);
   - две операции проверки эквивалентности (==) и (!=);
   - конкатенация или сцепление строк (+);
   - взятие индекса ([]).
4. Используя индексацию строк, выполните следующие задания:
   - из одной строки копировать содержание скобок комментария в новую строку (при условии того, что если скобки открываются, то они должны быть закрыты);
   - скопировать в новую строку все символы другой строки за исключением слова “student”;

Задание 2.

Массивы строк.

Подобно другим типам данных строки могут быть собраны в массивы.

1. Создайте консольное приложение ArrayOf-

Strings;

2. Объявите функцию SumInWords, которая должна получать параметр (целое число не менее четвертого порядка), а возвращать массив строк (с наименованием валюты).

Методические указания:

```
string[]  ar1 = {"один", "два",
                 "три", "четыре", "пять",
                 "шесть", "семь", "восемь",
                 "девять"};
```

И так несколько массивов для наименования валюты, десятков, сотен и т.д.

Например: Входящий параметр – 1908, результат – тысяча девятьсот восемь рублей;

Входящий параметр – 352, результат – триста пятьдесят два рубля;

Задание 3.

Методы Join и Split.

Методы Join и Split выполняют над строкой текста взаимно обратные преобразования. Динамический метод Split позволяет осуществить разбор текста на элементы. Статический метод Join выполняет обратную операцию, собирая строку из элементов.

Заданный строкой текст зачастую представляет собой совокупность структурированных элементов - абзацев, предложений, слов, скобочных выражений и т.д. При работе с таким текстом необходимо разделить его на элементы, пользуясь специальными разделителями элементов, - это могут быть пробелы, скобки, знаки препинания. Практически подобные задачи возникают постоянно при работе со структурированными текстами. Методы Split и Join облегчают решение этих задач.

Динамический ме- тод Split, как обычно, пере-

гружен. Наиболее часто используемая реализация имеет следующий синтаксис:

```
public string[] Split(params char[])
```

На вход методу Split передается один или несколько символов, интерпретируемых как разделители. Объект string, вызвавший метод, разделяется на подстроки, ограниченные этими разделителями. Из этих подстрок создается массив, возвращаемый в качестве результата метода. Другая реализация позволяет ограничить число элементов возвращаемого массива.

Синтаксис статического метода Join таков:

```
public static string Join(string delimiters, string[] items
)
```

В качестве результата метод возвращает строку, полученную конкатенацией элементов массива items, между которыми вставляется строка разделителей delimiters. Как правило, строка delimiters состоит из одного символа, который и разделяет в результирующей строке элементы массива items; но в отдельных случаях ограничителем может быть строка из нескольких символов.

1. Создайте консольное приложение SplitJoin;
2. Написать функциию, получающую в качестве параметра строку;
3. В этой функции объявите два массива строк:

```
string[] SimpleSentences, Words;
```

SimpleSentences – должен состоять из простых предложений, составляющих основную строку;

Words – должен состоять из слов, составляющих основную строку;

4. Используя встроенный метод класса String, разделить основную строку на простые предложения, в качестве разделителя использую ‘,’ и проинициализируйте массив

SimpleSentences:

```
SimpleSentences = txt.Split(',');
```

5. Аналогично разбейте строку на слова и инициализируйте ими второй массив;
6. Используя метод Join соберите строку из элементов массива Words, проанализируйте полученный результат. (Есть ли различия между исходной строкой и результатом? Если есть, то объясните их появление)

Задание 4.

Работа с объектами класса StringBuilder.

Класс string не разрешает изменять существующие объекты. Строковый класс StringBuilder позволяет компенсировать этот недостаток. Этот класс принадлежит к изменяемым классам и его можно найти в пространстве имен System.Text. Рассмотрим класс StringBuilder подробнее.

Объекты этого класса объявляются с явным вызовом конструктора класса. Поскольку специальных констант этого типа не существует, то вызов конструктора для инициализации объекта просто необходим. Конструктор класса перегружен, и наряду с конструктором без параметров, создающим пустую строку, имеется набор конструкторов, которым можно передать две группы параметров. Первая группа позволяет задать строку или подстроку, значением которой будет инициализироваться создаваемый объект класса StringBuilder. Вторая группа параметров позволяет задать емкость объекта - объем памяти, отводимой данному экземпляру класса StringBuilder. Каждая из этих групп не является обязательной и может быть опущена. Примером может служить конструктор без параметров, который создает объект, инициализированный пустой строкой, и с некоторой емкостью, заданной по умолчанию, значение которой зависит от реализации.

Над строками этого класса определены практи-

чески те же операции с той же семантикой, что и над строками класса String:

- присваивание (=);
- две операции проверки эквивалентности (==) и (!=);
- взятие индекса ([]).

Операция конкатенации (+) не определена над строками класса StringBuilder, ее роль играет метод Append, дописывающий новую строку в хвост уже существующей.

1. Создайте консольное приложение Example-StringBuilder;
2. Объявите функцию TestStringBuilder(), демонстрирующую главные отличия класса String-Builder от класса String:

Методические указания:

- объявите два объекта класса StringBuilder, используя конструктор:

StringBuilder str1=new StringBuilder("String – example of StringBuilder");

- инициализируйте оба объекта;
- приведите примеры изменения данных объектов;

3. Вызовите эту функция через функцию Main, проанализируйте полученные результаты;

У класса StringBuilder методов значительно меньше, чем у класса String, т.к. StringBuilder дает возможность изменять строки. По этой причине у класса есть основные методы, позволяющие выполнять такие операции над строкой как вставка, удаление и замена подстрок, но нет методов, подобных поиску вхождения, которые можно выполнять над обычными строками. Технология работы обычно такова: конструируется строка класса StringBuilder; выполняются операции, требующие изменение значения; полученная строка преобразуется в строку класса String; над

этой строкой выполняются операции, не требующие изменения значения строки.

Основные методы StringBuilder:

public StringBuilder Append (<объект>)
к строке, вызвавшей метод, присоединяется строка, полученная из объекта, который передан методу в качестве параметра;
public StringBuilder Insert (int location,<объект>)
вставляет строку, полученную из объекта, в позицию, указанную параметром location.
public StringBuilder Remove (int start, int len)
удаляет подстроку длины len, начинающуюся с позиции start;
public StringBuilder Replace (string str1,string str2)
все вхождения подстроки str1 заменяются на строку str2;
public StringBuilder AppendFormat (<строка форматов>, <объекты>)
Метод является комбинацией метода Format класса String и метода Append. Строка форматов, переданная методу, содержит только спецификации форматов. В соответствии с этими спецификациями находятся и форматируются объекты. Полученные в результате форматирования строки присоединяются в конец исходной строки.

За исключением метода Remove, все рассмотренные методы являются перегруженными. В их описании дана схема вызова метода, а не точный синтаксис перегруженных реализаций. Приведу примеры, чтобы продемонстрировать, как вызываются и как работают эти методы:

4. В функции Main сконструируйте строку, используя методы Insert и Append.
5. Затем сконструируйте строку. Выведите ее на

экран пронумеровав, каждое простое предложение входящее в эту строку.

Методические указания:

- в цикле foreach используйте метод AppendFormat (в качестве разделителя простых предложений в составе сложного примите ‘,’):
txtbuild.AppendFormat(" {0}: {1} ", num++,sub);

6. Выведите на экран обе строки. Полученные результаты проанализируйте.

Задание 5.

Емкость буфера.

Каждый экземпляр строки класса StringBuilder имеет буфер, в котором хранится строка. Объем буфера - его емкость - может меняться в процессе работы со строкой. Объекты класса имеют две характеристики емкости - текущую и максимальную. В процессе работы текущая емкость изменяется, естественно, в пределах максимальной емкости, которая реально достаточно высока. У класса StringBuilder имеется 2 свойства и один метод, позволяющие анализировать и управлять емкостными свойствами буфера.

- свойство Capacity - возвращает или устанавливает текущую емкость буфера;
- свойство MaxCapacity - возвращает максимальную емкость буфера. Результат один и тот же для всех экземпляров класса;
- метод int EnsureCapacity (int capacity) - позволяет уменьшить емкость буфера. Метод пытается вначале установить емкость, заданную параметром capacity; если это значение меньше размера хранимой строки, то емкость устанавливается такой, чтобы гарантировать размещение строки. Это

число и возвращается в качестве результата работы метода. (Объясните в каких случаях возникает потребность в уменьшении емкости буфера???)

1. Откройте консольное приложение из предыдущего задания;
2. С помощью методов Capacity и Maxcapacity измерьте текущие и максимальные емкости буфера соответственно для txtbuild и strbuild;
3. Выведите на экран полученные значения емкостей, проанализируйте полученный результат;
4. С помощью метода EnsureCapacity уменьшить емкость буфера для обоих объектов до определенного значения, и также произведите вывод на экран новых значений емкостей (объяснить разницу полученных значений);

Задание 6.

Массив символов char[].

В языке C# определен класс Char[], и его можно использовать для представления строк постоянной длины, как это делается в C++. Более того, поскольку массивы в C# динамические, то расширяется класс задач, в которых можно использовать массивы символов для представления строк. Так что имеет смысл разобраться, насколько хорошо C# поддерживает работу с таким представлением строк.

Массив char[] - это обычный массив. Его нельзя инициализировать строкой символов, как это разрешается в C++. Константа, задающая строку символов, принадлежит классу String, а в C# не определены взаимные преобразования между классами String и Char[], даже явные. У класса String есть, правда, динамический метод ToCharArray, задающий подобное преобразование. Возможно также, посим-

вольно передать содержимое переменной string в массив символов.

Метод ToCharArray позволяет преобразовать строку в массив символов. К сожалению, обратная операция не определена, поскольку метод ToString, которым, конечно же, обладают все объекты класса Char[], печатает информацию о классе, а не содержимое массива.

Класс Char[], как и всякий класс-массив в C#, является наследником не только класса Object, но и класса Array, и, следовательно, обладает всеми методами родительских классов. А есть ли у него специфические методы, которые позволяют выполнять операции над строками, представленными массивами символов? Таких специальных операций нет. Но некоторые перегруженные методы класса Array можно рассматривать как операции над строками. Например, метод Copy дает возможность выделять и заменять подстроку в теле строки. Методы IndexOf, LastIndexOf позволяют определить индексы первого и последнего вхождения в строку некоторого символа.

1. Создайте консольное приложение CharArray;
2. Создайте несколько строк, инициализируя их различными способами;
   Методические указания:
   ```
   char[] str1 = "Hello, World!";
   string str2 = "Здравствуй, Мир!";
   ```
   В каком случае выведется сообщение об ошибке и почему?
3. Напишите функцию PrintCharAr(), выводящую на экран массив символов, получаемый в качестве входящего параметра;
   Методические указания:
   В цикле произвести вывод посимвольно:
   ```
   Console.Write(ar[i]);
   ```
4. Напишите функцию CharArrayToString(), об-

ратную методу ToCharArray, т.е. преобразующую массив символов в строку;

Методические указания:

```
String CharArrayToString(char[] ar)
//функция возвращает объект типа String
```

5. Напишите функцию IndexOfStr, находящую индекс вхождения подстроки в строку:

   Методически указания:

```
int IndexOfStr( char[]s1, char[] s2) //
нахождение строки s2 в s1
```

6. Напишите процедуру TestIndexSym(), определяющую индексы вхождения символов и подстрок в строку;
7. Проанализируйте результаты выполнения всех функций.

Контрольные вопросы:

1. Почему нельзя присвоить значение символу внутри строки, используя индекс?
2. Почему оператор сравнения == работает с типом String как со значимыми типами данных?
3. Перечислите методы класса String;
4. В чем заключается различие между классами String, StringBuilder и массивами символов?
5. Перечислите преимущества и недостатки класса StringBuilder.
6. Перечислите методы класса StringBuilder;
7. Объясните процесс инициализации символьного массива строкой.
8. Объясните работу и назначение свойств класса StringBuilder Capacity и MaxCapacity.

ЛАБОРАТОРНАЯ РАБОТА №6. ПАРАМЕТРЫ МЕТОДОВ

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

Синтаксис объявления метода:

[атрибуты] [спецификаторы] <тип возвращаемого значения> имя_метода([параметры])

Параметры используется для обмена информацией с методом. Параметр представляет собой локальную переменную, которая при вызове метода принимает значение соответствующего аргумента. Область действия параметра – весь метод. 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 попарно сопоставляется соответствующий аргумент.
4. Выполняется тело метода
5. Если метод возвращает значение, оно передаётся в точку вызова.

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 аргумента. Параметр-массив может быть только один и должен располагаться последним в списке.

```
public int Calculate(
  int a,
  ref int b,
  out int c,
```

```
params int[] d)
```

Задание 1. Параметры-значения.

В отличие от ссылок на объекты, переменные со значимыми типами передаются в функцию по значению, то есть функции передается значение, содержащееся в этой переменной, но не сама переменная. При такой передаче изменение значения соответствующей переменной внутри функции не вызовет изменения значения переданной переменной в вызывающей программе.

1. Создайте новое консольное приложение.
2. Добавьте в проект класс Example, методы которого будут использоваться для примера передачи параметров-значений.
    a. Создайте открытый статический метод Sqr, принимающий в качестве параметров два значения типа double и не возвращающий значения. Этот метод будет возводить два числа в квадрат и выводить результат на экран (для этого используется статический метод Pow класса Math).

    Методические указания.

```
{
    a = Math.Pow(a, 2);
        b = Math.Pow(b, 2);
    Console.WriteLine("В методе Sqr. a =
{0}\tb = {1}", a, b);
}
```

    b. В методе Main создайте и инициализируйте две переменных типа double. Выведите на консоль значения переменных, затем вызовите метод Sqr, указав их в качестве аргументов, после чего повторите вывод на экран.
    c. Теперь снова вызовите метод Sqr, в параметрах указав удвоенные исходные переменные. Покажите значения переменных в конце функции Main.

    Методические указания.

```
Example.Sqr(x * 2, y * 2);
Console.WriteLine("В методе Main: a
```

```
= {0}\tb = {1}", a, b);
```

d. Скомпилируйте программу и посмотрите результаты работы

Как мы ни старались, изменяя значения параметров внутри функции или при передаче их в функцию, присвоить им новые значения не удалось. Дело в том, что 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.

3. Для примера передачи параметров ссылочного типа воспользуемся уже существующим классом Example, дополнив его:
   a. Объявите в классе два открытых поля типа int.
   b. Создайте открытый конструктор, принимающий один параметр целочисленного типа и инициализирующий одно из полей этим значением.

      Методические указания.

```
public int x;
public int y;
public Example(int x)
{
 this.x = x;
}
```

   c. Добавьте в класс Example статический метод с именем Init, принимающий в качестве параметра экземпляр этого же класса и не возвращающий ничего в вызывающую функцию. Метод должен присваивать обоим полям объекта-параметра значения, равные 1. В теле метода так же предусмотрите вывод на консоль значений полей объекта-параметра после их изменения.
   d. В методе Main создайте объект класса Example, передав его конструктору неединичное значение;
   e. Выведите на консоль значения полей созданного объекта;

f. Передайте только что созданный объект статическому методу Init после чего посмотрите значения его полей с помощью метода Console.WriteLine();

Методические указания.

```
Console.WriteLine("В методе Main: Obj.x = {0}\tObj.y = {1}", Obj.x, Obj.y);
Example.Init(Obj);
Console.WriteLine("В методе Main: Obj.x = {0}\tObj.y = {1}", Obj.x, Obj.y);
```

g. Скомпилируйте измененную программу и посмотрите результат её работы.

Значения полей объекта изменились, несмотря на то, что он был передан функции по значению. Причина в том, что переменная-объект на самом деле хранит ссылку на данные, расположенные в динамической памяти, и именно эта ссылка передаётся в метод. Метод получает в своё распоряжение фактический адрес данных и, следовательно, может их изменить.

4. Измените метод Init таким образом, чтобы он создавал новый объект класса Example, и присваивал его передаваемому в метод параметру. После этого, как и раньше, полям объекта должны присваиваться новые значения, которые будут выведены на консоль в теле метода.

Задание 2. Параметры-ссылки

Если в методе требуется изменить значение каких-либо передаваемых в него величин, используются параметры-ссылки. Признаком параметра-сслыки является ключевое слово ref перед описанием параметра:

```
ref int x
```

При вызове метода в область параметров копируется не значение аргумента, а его адрес, и метод через него имеет доступ к ячейке, в которой хранится аргумент. Метод работает непосред-

ственно с переменной из вызывающей функции и, следовательно, может её изменить. При вызове метода перед именем параметра указывается ключевое слово ref.

1. Добавьте в решение (Solution) из первого задания новый проект консольного приложения.
   a. В меню выберите пункт File->Add->Create project.
   b. Далее в окне выбора проекта укажите тип проекта Console Application.
   c. Укажите имя проекта и нажмите OK.
2. Скопируйте код класса Example и функции Main из первого проекта и вставьте его во второй.
3. В методе Sqr измените объявление так, чтобы оба параметра передавались в метод по ссылке.

   Методические указания.

```
public static void Sqr(ref double a, ref double b)
```

4. В функции Init так же измените тип параметра с параметра-значения на параметр-ссылку.
5. Скомпилируйте проект.
   a. При необходимости, измените код функции Main.
   b. Если какой-либо из вызовов функций не корректен для параметров-ссылок, закомментируйте его.
   c. Посмотрите результаты работы программы.

6. Вызовите метод Sqr с неинициализированной переменной типа double.
   a. В теле функции Main объявите, но не инициализируйте новую переменную нужного типа.
   b. Добавьте вызов метода Sqr, передав в качестве второго аргумента только что объявленную переменную.

      Методические указания.

```
double c;
Example.Sqr(ref a, ref c);
```

   c. Попробуйте скомпилировать проект.

Задание 3. Выходные параметры.

Довольно часто возникает необходимость в методах, которые формируют несколько величин, например, если в методе создаются объекты или инициализируются ресурсы. В этом случае становится неудобным ограничение параметров-ссылок: необходимость присваивания значения аргументу до вызова метода. Это ограничение снимает спецификатор out. Параметру, имеющему этот спецификатор, должно быть обязательно присвоено значение внутри метода, компилятор за этим следит. Зато в вызывающем коде можно ограничиться описанием переменной без инициализации.

3. Добавьте в существующее решение новый консольный проект так же, как в задании 2.
4. Создайте статический метод, принимающий параметр вещественного типа и возвращающий информацию о нём: знак числа, является ли число целым, модуль числа, а так же квадрат числа. При этом само число по выходу из функции измениться не должно.
    - Объявите функцию с одним параметром-значением типа double и выходными параметрами нужных значений: int для знака числа, bool для проверки на целочисленность, double для модуля числа и для квадрата числа. Функция не возвращает значения.
    - В теле функции присвойте соответствующим параметрам необходимые значения в зависимости от входного параметра. Для этой цели удобно использовать статические методы класса Math из пространства имён System.

      Методические указания.

```
sign = Math.Sign(value);
isInteger = value % 1 == 0 ? true :
false;
abs = Math.Abs(value);
sqr = value * value;
```

5. В функции Main создайте две переменных типа double и инициализируйте их значениями 42 и -12.67.
6. Объявите, но не инициализируйте ещё две переменных типа double и по одной переменной типа int и bool.
7. Дважды вызовите метод, формирующий

информацию о числе с различными инициализированными переменными. Не забудьте указать в качестве выходных параметров переменные нужного типа. После каждого вызова выводите на консоль результаты работы функции.

8. Скомпилируйте программу и посмотрите на результат её работы.

Задание 4. Параметры-массивы.

Иногда бывает удобно создать метод, в который можно передавать разное число аргументов. Язык C# предоставляет такую возможность с помощью ключевого слова params. Параметр, помеченный этим ключевым словом, может быть только один и должен располагаться последним в списке. Он обозначает массив заданного типа неопределенной длины. Внутри метода к параметрам обращаются, как к обычным элементам массива. Параметр-массив поддерживает все свойства и методы, характерные для массивов.

1. Добавьте четвертый проект консольного приложения к уже сушествующим.
2. Создайте статический метод Avarge, принимающий параметр-массив arr целочисленного типа и возвращающий среднее значение элементов массива.

Методические указания.

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

3. В функции Main определите массив типа int, состоящий из 4 элементов. С помощью функции Avarge вычислите среднее значение элементов массива.
4. Объявите и инициализируйте две переменные типа short и одну переменную типа byte. Передайте их в качестве параметров функции Avarge.

Методические указания.

```
short z = 1, e = 12;
byte v = 107;
Console.WriteLine(Avarge(z, e, v));
```

5. Вызовите метод Avarge без параметров.
6. Скомпилируйте проект.

Задание 5. Узкие места передачи параметров в функцию.

1. Создайте проекты, иллюстрирующие следующие особенности передачи параметров в функцию.
    a. Отличие параметров <тип>[] от params <тип>[]. Передача массивов в функцию.
    b. params – по значению или по ссылке.
    c. Передача в методы строк. Изменение строк в методе.
    d. Object как формальный параметр функции. Определение типа переданного в функцию аргумента.

Контрольные вопросы

5) Объясните результат работы последнего пункта программы, иллюстрирующей передачу параметров-значений.
6) Чем отличается передача аргумента ссылочного типа по значению от передачи по ссылке?
7) Почему при передаче в метод в качестве параметра-ссылки неинициализированного значения, проект не компилируется?
8) В чём отличие между параметрами-ссылками и выходными параметрами методов?
9) Можно ли в качестве выходного параметра передавать в функцию инициализированную переменную?
10) Аргументы каких типов можно передавать в функцию со следующим объявлением:
```
int Example(params double[] x)?
```
11) Можно ли изменить параметр типа string в теле функции? Изменится ли исходная строка после возврата в точку вызова?

12) Какие модификаторы (ключевые слова) могут использоваться для передачи параметров в функцию?
13) Возможно ли передать один и тот же параметр по ссылке в одну функцию несколько раз? А по значению?
14) Можно ли передавать ссылочный тип с модификатором ref?
15) Необходимо ли инициализировать переменную, которая будет переданна в функцию с модификатором out?
16) Что случиться, если не изменить значение переменной, переданной в функцию с модификатором out?
17) Какое из объявлений функции являеться верным:

```
static void MyFunction(int a,params int[] values);
```

или

```
static void MyFunction(params double[] values, int a);
```

?

ЛАБОРАТОРНАЯ РАБОТА №7

Тема: Определение классов, статические конструкторы, конструкторы, методы-свойства.

Задание 1. Методы-свойства.

Методы, называемые свойствами (Properties), представляют специальную синтаксическую конструкцию, предназначенную для обеспечения эффективной работы со свойствами. При работе со свойствами объекта (полями) часто нужно решить, какой модификатор доступа использовать, чтобы реализовать нужную стратегию доступа к полю класса.

5 наиболее употребительных стратегий:

18) чтение, запись (Read, Write);
19) чтение, запись при первом обращении (Read, Write-once);
20) только чтение (Read-only);
21) только запись (Write-only);
22) ни чтения, ни записи (Not Read, Not Write).

Открытость свойств (атрибут public) позволяет реализовать только первую стратегию. В языке C# принято, как и в других объектных языках, свойства объявлять закрытыми, а нужную стратегию доступа организовывать через методы. Для эффективности этого процесса и введены специальные методы-свойства.

2. В Visual Studio создайте проект консольного приложения. (C#)
3. Создайте класс с именем Person, у которого пять полей: fam, status, salary, age, health, характеризующих соответственно фамилию, статус, зарплату, возраст и здоровье персоны.

Для каждого из этих полей может быть разумной своя стратегия доступа. Возраст доступен для чтения и записи, фамилию можно задать только один раз, статус можно только читать, зарплата недоступна для чтения, а здоровье закрыто для доступа и только специальные методы класса могут сообщать некоторую информацию о здоровье персоны.

4. Обеспечьте доступ к закрытому полю fam класса Person следующим образом:

```
// стратегия: Read,Write-once (Чтение, запись при первом обращении)
public string Fam
  {
    set {if (fam == "") fam = value;}
    get {return(fam);}
  }
```

Метод get возвращает значение закрытого поля, метод set - устанавливает значение, используя передаваемое ему значение в момент вызова, хранящееся в служебной переменной со стандартным именем value.

5. Реализуйте стратегии доступа к остальным полям класса Person.
6. В методе Main консольного приложения напишите пример, показывающий создание и работу с полями персоны.
7. Запустите консольное приложение, чтобы убедиться, что оно работает корректно.

Задание №2. Проектирование класса Rational,

описывающего рациональные числа.

Класс Rational определяет новый тип данных - рациональные числа и основные операции над ними - сложение, умножение, вычитание и деление. Рациональное число задается парой целых чисел (m,n) и изображается обычно в виде дроби m/n. Число m называется числителем, n - знаменателем. Для каждого рационального числа существует множество его представлений, например, 1/2, 2/4, 3/6, 6/12 - задают одно и тоже рациональное число. Среди всех представлений можно выделить то, в котором числитель и знаменатель взаимно несократимы. Такой представитель будет храниться в полях класса. Операции над рациональными числами определяются естественным для математики образом

4) В Visual Studio создайте проект консольного приложения. (C#)
5) Создайте класс с именем Rational.
6) В классе Rational определите закрытые поля:
   m-числитель
   n-знаменатель

7) Определите метод, вычисляющий наибольший общий делитель пары чисел. **Этот метод понадобится не только конструктору класса, но и всем операциям над рациональными числами.**
8) Определите конструктор с аргументами, которому будут передаваться два целых: числитель и знаменатель создаваемого числа

Методические указания:

Конструктор класса. Создает рациональное число m/n, эквивалентное a/b, но с взаимно несократимыми числителем и знаменателем. Если b=0, то результатом является рациональное число 0 -пара (0,1).

9) Переопределите метод ToString();
10) Напишите функцию PrintRational(), позволяющую выводить на печать данные о классе.
11) Для проверки корректности создания рациональных чисел

в методе Main консольного приложения напишите код, который создает и печатает несколько рациональных чисел.

12) Запустите консольное приложение, чтобы убедиться, что оно работает корректно.

Задание №3. Операции над рациональными числами

Определим над рациональными числами стандартный набор операций - сложение и вычитание, умножение и деление. Реализуем эти операции методами с именами Plus, Minus, Mult, Divide соответственно. Поскольку рациональные числа - это, прежде всего именно числа, то для выполнения операций над ними часто удобнее пользоваться привычными знаками операций (+, -, *, /). Язык C# допускает определение операций, заданных указанными символами. Этот процесс называется **перегрузкой операций**.

7. Откройте консольное приложение созданное в задании 1.
8. Реализуйте Метод Plus.

Методические указания:

По правилам сложения дробей вычисляется числитель и знаменатель результата, и эти данные становятся аргументами конструктора, создающего требуемое рациональное число, которое удовлетворяет правилам класса.

9. Перегрузите операцию «+» класса Rational.

Методические указания:

Именем соответствующего метода является сам знак операции, которому предшествует ключевое слово operator. Важно также помнить, что операция является статическим методом класса с атрибутом static.

10. Аналогичным образом реализуйте методы Minus, Mult, Divide и перегрузите операции «-», «*», «/»над рациональными числами.
11. Проверьте работу всех операций, вызывая метод или операцию в зависимости от конкретной ситуации.

Обратите внимание: при перегрузке операций сохраняется общепринятый приоритет операций. Поэтому при вычислении выражения r3+r4*r5 вначале будет выполняться умножение рациональных чисел, а потом уже сложение.

Задание №4. Константы класса Rational.

Рассмотрим важную проблему определения констант в собственном классе. Определим две константы 0 и 1 класса Rational. Кажется, что сделать это невозможно из-за ограничений, накладываемых на объявление констант. Напомню, константы должны быть инициализированы в момент объявления, и их значения должны быть заданы константными выражениями, известными в момент компиляции. Но в момент компиляции у класса Rational нет никаких известных константных выражений. Как же быть? Справиться с проблемой поможет **статический конструктор**, созданный для решения подобных задач. Роль констант класса будут играть статические поля, объявленные с атрибутом readonly, то есть доступные только для чтения. Нам также будет полезен закрытый конструктор класса. Еще укажем, что введение констант класса требует использования экзотических средств языка C#.

9. Откройте консольное приложение созданное в 2-х предыдущих заданиях.

10. Определите закрытый конструктор:

```
private Rational(int a, int b, string t)
{
  m = a; n = b;
}
```

При перегрузке методов (в данном случае конструкторов) сигнатуры должны различаться, и поэтому пришлось ввести дополнительный аргумент t для избежания конфликтов. Поскольку конструктор закрытый, то гарантируется корректное задание аргументов при его вызове.

11. Определите константы Zero и One класса Rational, которые задаются статическими полями с атрибутом

readonly.

12. Задайте статический конструктор, в котором определяются значения констант: static Rational()
```
{
  …
}
```

13. Добавьте в класс булевы операции над рациональными числами - равенство и неравенство, больше и меньше. При этом две последние операции сделайте перегруженными, позволяя сравнивать рациональные числа с числами типа double.
14. Создайте функцию TestRational(), демонстрирующую работу с константами, булевыми и арифметическими выражениями над рациональными числами.
15. Запустите консольное приложение, чтобы убедиться, что оно работает корректно.

Контрольные вопросы:

- Как вызываются статические поля и методы?
- Как и когда происходит создание объектов?
- Зачем классу нужно несколько конструкторов?
- Для чего нужен статический конструктор? Для чего нужен закрытый конструктор?
- Что понимается под перегрузкой операции?
- Назовите методы, которые используются для работы со свойствами?

7. Зачем необходим аргумент t при объявлении конструктора в 4 задании:
```
private Rational(int a, int b, string t)
{
  m = a; n = b;
}
```

СПИСОК ИССПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Язык программирования C#, Андерс Хейлсберг, СПБ, Издетельский дом «Питер», 2012

2. Программирование на языке высокого уровня C#, www.intuit.ru

3. Объектное программирование в классах на C#, www.intuit.ru