Урок № 2

Содержание

1. Массивы

1.1. Одномерные массивы.

Массив в C#, как и в C – это группа ячеек памяти, имеющих одно имя и тип данных.

Программа может обращаться к любому элементу массива путем указания имени массива, за которым в квадратных скобках следует индекс (значение, указывающее на местоположение элемента в рамках массива) элемента. Если в качестве индекса программа использует целочисленное выражение, тогда сначала это выражение вычисляется для определения индекса.

Согласно общеязыковой спецификации (CLS), нумерация элементов в массиве должна начинаться с нуля. Если следовать этому правилу, тогда методы, написанные на C#, смогут передать ссылку на созданный массив коду, написанному на другом языке, например, на Microsoft Visual Basic .NET. Кроме того, Microsoft постаралась оптимизировать работу массивов с начальным нулевым индексом, поскольку они получили очень большое распространение. Тем не менее, в CLR допускаются и иные варианты индексации массивов, хотя это не приветствуется.

Синтаксис объявления одномерного массива следующий:

<Тип элементов массива>[] <имя массива>;

Мы видим, что данный синтаксис отличается от привычного вам «сишного». Квадратные скобки перекочевали от имени переменной – к имени типа, явно подчеркивая таким образом, что переменная имеет «массивный тип».

Примеры объявления одномерных массивов:

```
int[] myArr;            // Объявление ссылки на массив целых чисел
string [] myStrings;    // Объявление ссылки на массив строк
```

Все массивы в C# унаследованы от класса `System.Array`, который, в свою очередь, наследуется от класса `System.Object`. Это наследование означает, что все массивы являются объектами (к чему теперь придется привыкать). «Объектность» массивов с одной стороны дает ряд преимуществ, с другой – имеет ряд недостатков. К преимуществам можно отнести: полученный в наследство немалый набор методов по работе с массивами, контроль выхода за границы массива и др.; к недостаткам –

некоторое снижение быстродействия при работе с массивом вследствие того, что он размещается в «куче».

Учитывая, что массивы – это ссылочные типы, в приведенном выше примере создаются две пустые ссылки. Для дальнейшей работы необходимо выделить память под эти ссылки. Как вы уже и привыкли, память выделяется с помощью «*new*».

```
myArr = new int[10];          // Выделяем память под массив на 10 чисел
myStrings = new string[50];   // Выделяем память под массив на 50 строк
```

После выделения памяти инициализация элементов происходит следующим образом: значения всех простых типов устанавливаются в «*0*», значения логического типа – в *false*, ссылки – в *null*.

Есть также возможность проинициализировать массив нужными значениями при объявлении:

```
int[] myArr = new int[10] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int[] myArr = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int[] myArr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

В первом случае, если количество элементов в списке инициализации окажется больше или меньше чем заказано – то компилятор выдаст сообщение об ошибке. Т.е. вариант «сишного» обнуления недостающих элементов здесь не пройдет (да и не надо, ведь все элементы и так проинициализируются нулями). Во втором и третьем случае размер массива вычисляется из количества элементов в списке инициализации.

Для прохода по массиву можно использовать любой из циклов *for*, *while*, *do-while*, *foreach*. Например, заполнение массива можно было бы сделать следующим образом:

```
int[] myArr = new int[10];
for (int i = 0; i < myArray.Length; ++i)
      myArray[i] = i + 1;
```

Так как все массивы являются объектами, то в них помимо данных-значений хранится и дополнительная информация, как то размер, нижняя граница, тип массива, количество измерений, количество элементов в каждом измерении и др. Размещение массива в памяти можно схематически представить так:

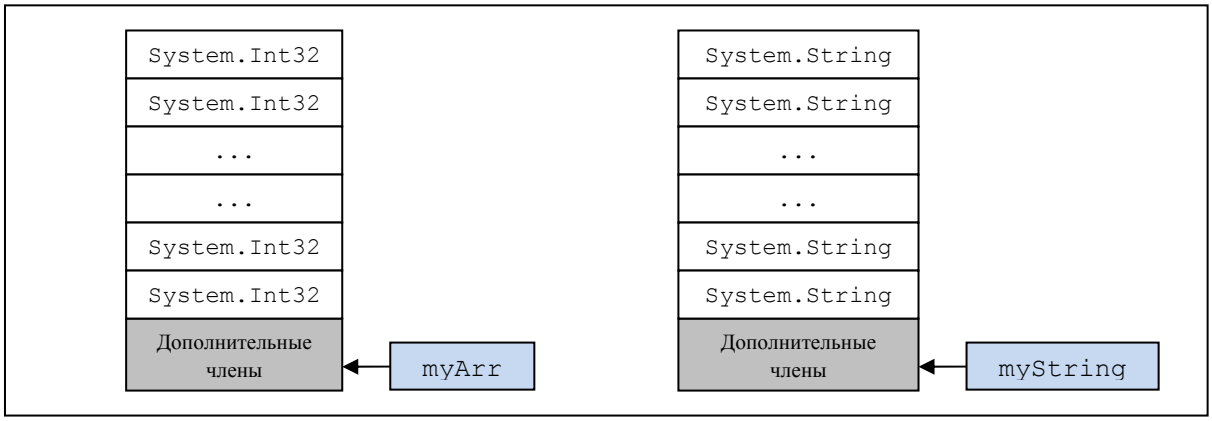


Рис. 1.1. Схема размещения массива в памяти

1.2. Многомерные массивы.

Одномерные массивы (они же векторы) обеспечивают наилучшую производительность, за счет того, что с ними используются специальные команды IL языка. Но в случае необходимости можно использовать и многомерные массивы. Синтаксис многомерного массива еще больше отличается от «сишного» и более похож на «паскалевский».

<Тип элементов массива>[, ,...] <имя массива>;

```
float[,] myArr;          // Объявление ссылки на двумерный массив дробных
                         // чисел
myArr = new float[2, 3];// Выделение памяти под ссылку на двумерный
                         // массив дробных чисел
```

Инициализация двумерного массива при объявлении:

```
int[,] myArr = new int[2, 3] { { 1, 2, 3 }, { 3, 4, 6 } };
int[,] myArr = new int[,] { { 1, 2, 3 }, { 3, 4, 6 } };
int[,] myArr = { { 1, 2, 3 }, { 3, 4, 6 } };
```

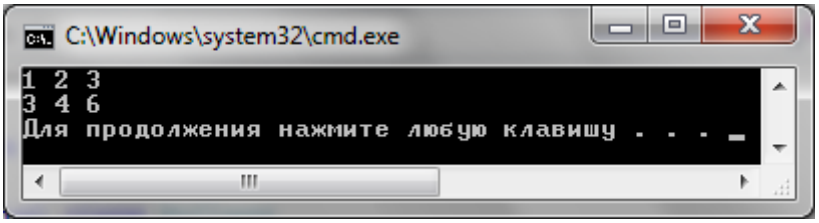
Доступ к элементам двумерного массива осуществляется следующим образом:

```
myArr[1, 2] = 100;                  // Занесение значения в третью
                                    // колонку второй строки
Console.WriteLine(myArr[0, 0]);     // Вывод на экран элемента первой
                                    //колонки в первой строке
```

Пример вывода двухмерного массива на экран:

```
int[,] myArr = { { 1, 2, 3 }, { 3, 4, 6 } };
for (int i = 0; i < myArr.GetLength(0); ++i)
{
      for (int j = 0; j < myArr.GetLength(1); ++j)
            Console.Write(myArr[i, j] + " ");
      Console.WriteLine();
}
```

Функию `GetLength` будем рассматривать в следующем разделе, а пока просто поверье, что она возвращает длину указанного измерения массива. Результат выполнения:



1.3. **Рваные массивы.**

Кроме одномерных и многомерных массивов C# также поддерживает «рваные» (или вложенные) массивы. Синтаксис объявления такого массива снова отличается от предыдущих, но это будет последний тип массивов, синтаксис которого вам придется выучить.

Итак, синтаксис объявления рваного массива выглядит так:

<Тип элементов массива>[][] <имя массива>;

Такой массив представляет собой массив массивов. Т.е. в каждой ячейке массива располагается вектор. За счет того, что при работе с вложенными массивами мы фактически работаем с векторами – то производительность такая же, как и при работе с обычными одномерными. При этом нужно учитывать, что все «подмассивы» не находятся в памяти рядом.

```
int[][] myArr = new int[2][]; // Создание внешнего массива на две ячейки
myArr[0] = new int[] {1, 2};  // Создание внутреннего вектора в первой
                              // ячейке
myArr[1] = new int[] {3, 4, 5, 6};  // Создание внутреннего вектора во
                                    // второй ячейке
```

Итого получаем следующий массив:

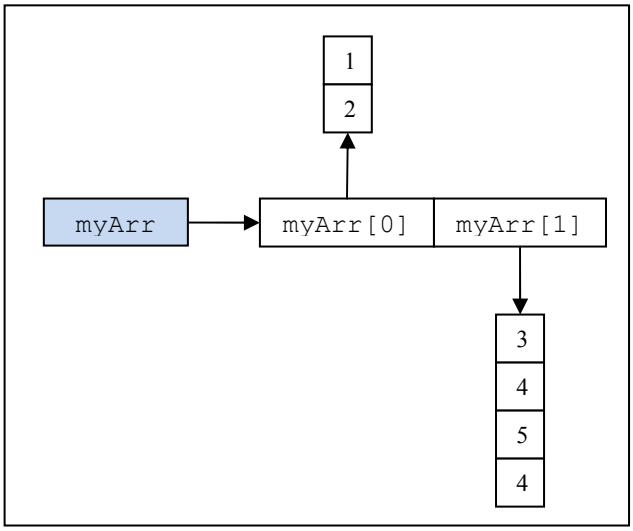


Рис. 1.2. Схема рваного массива

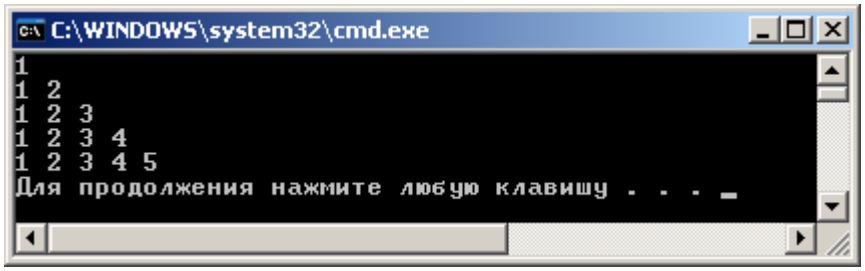
Доступ к элементам такого массива получаем следующим образом:

```
Console.WriteLine(myArr[1][2]);     // На экране увидим «5»
```

Поскольку работа с данным видом массивов может вызывать трудности – рассмотрим пример, где осуществляется заполнение массива и вывод на экран:

```
static void Main(string[] args)
{
      int size = 5;
      // Объявление вложеного массива
      int[][] myArr = new int[size][];
      for (int i = 0; i < size; ++i)
      {
            // Создание внутренних массивов
            myArr[i] = new int[i + 1];
            for (int j = 0; j < i + 1; ++j)
            {
                  // Заполнение внутренних массивов
                  myArr[i][j] = j + 1;
                  // Вывод на экран элементов
                  Console.Write(myArr[i][j] + " ");
            }
            Console.WriteLine();
      }
}
```

Результаты выполнения:



Поскольку все массивы унаследованы от класса `System.Array`, то все они имеют богатый набор методов. Давайте рассмотрим некоторые полезные методы, полученные в «подарок» от родительского класса:

- Метод **`GetLength`** возвращает количество элементов массива по заданному измерению;
- Методы **`GetLowerBound`** и **`GetUpperBound`** возвращают соответственно нижнюю и верхнюю границы массива по заданному измерению (например, если есть одномерный массив на 5 элементов, то нижняя граница будет «`0`», верхняя – «`4`»)
- Метод **`CopyTo`** копирует все элементы одного одномерного массива в другой, начиная с заданной позиции.
- Метод **`Clone`** производит поверхностное копирование массива. Копия возвращается в виде массива **`System.Object[]`**
- Статический метод **`BinarySearch`** производит бинарный поиск значения в массиве (в диапазоне массива)
- Статический метод **`Clear`** очищает массив (диапазон массива). При этом ссылочные элементы устанавливаются в **`null`**, логические – в **`false`**, остальные типы значений – в «`0`»
- Статический метод **`IndexOf`** – возвращает индекс первого вхождения искомого элемента в массиве (в диапазоне массива), в случае неудачи – возвращает «`-1`». Поиск производится от начала массива.
- Статический метод **`LastIndexOf`** – возвращает индекс первого вхождения искомого элемента в массиве (в диапазоне массива). Поиск производится с конца массива, в случае неудачи – возвращает «`-1`».
- Статический метод **`Resize`** изменяет размер массива.
- Статический метод **`Reverse`** – реверсирует массив (диапазон массива).
- Статический метод **`Sort`** – сортирует массив (диапазон массива).

Также присутствуют методы расширения:

- Метод **`Sum`** – суммирует элементы массива
- Метод **`Average`** – подсчитывает среднее арифметическое элементов массива.

- Метод **`Contains`** – возвращает истину, если заданный элемент присутствует в массиве.
- Метод **`Max`** – возвращает максимальный элемент массива.
- Метод **`Min`** – возвращает минимальный элемент массива.

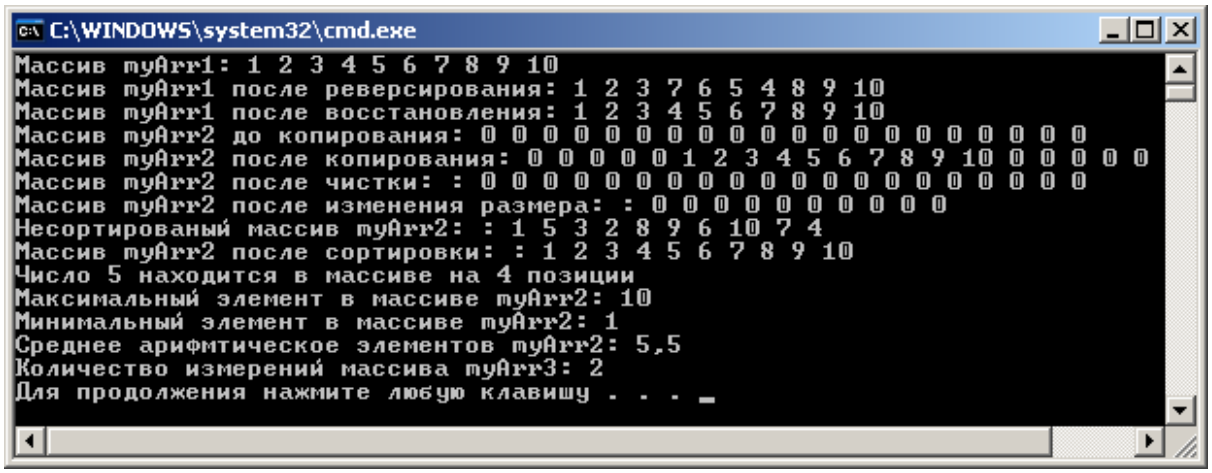
И на последок пара свойств:

- Свойство **`Length`** – возвращает длину массива
- Свойство **`Rank`** – возвращает количество измерений массива.

А теперь, чтоб закрепить свойства и методы массивов – рассмотрим следующий пример:

```
static void Main(string[] args)
{
      int[] myArr1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
      PrintArr("Массив myArr1", myArr1);
      int[] tempArr = (int[])myArr1.Clone();
      Array.Reverse(myArr1, 3, 4);
      PrintArr("Массив myArr1 после реверсирования", myArr1);
      myArr1 = tempArr;
      PrintArr("Массив myArr1 после восстановления", myArr1);

      int[] myArr2 = new int[20];
      PrintArr("Массив myArr2 до копирования", myArr2);
      myArr1.CopyTo(myArr2, 5);
      PrintArr("Массив myArr2 после копирования", myArr2);
      Array.Clear(myArr2, 0, myArr2.GetLength(0));
      PrintArr("Массив myArr2 после чистки: ", myArr2);
      Array.Resize(ref myArr2, 10);
      PrintArr("Массив myArr2 после изменения размера: ", myArr2);
      myArr2 = new[] { 1, 5, 3, 2, 8, 9, 6, 10, 7, 4 };
      PrintArr("Несортированый массив myArr2: ", myArr2);
      Array.Sort(myArr2);
      PrintArr("Массив myArr2 после сортировки: ", myArr2);
      Console.WriteLine("Число 5 находится в массиве на " +
      Array.BinarySearch(myArr2, 5) + " позиции");
      Console.WriteLine("Максимальный элемент в массиве myArr2: " +
            myArr2.Max());
      Console.WriteLine("Минимальный элемент в массиве myArr2: " +
            myArr2.Min());
      Console.WriteLine("Среднее арифмтическое элементов myArr2: " +
            myArr2.Average());

      int[,] myArr3 = { { 1, 2, 3 }, { 4, 5, 6 } };
      Console.WriteLine("Количество измерений массива myArr3: " +
      myArr3.Rank);
}

static void PrintArr(string text, int[] arr)
{
      Console.Write(text + ": ");
      for (int i = 0; i < arr.Length; ++i)
          Console.Write(arr[i] + " ");
      Console.WriteLine();
}
```

Результаты выполнения:



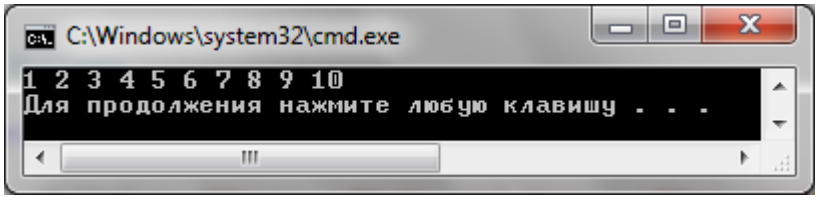
В примере иллюстрируется работа с вышеперечисленными функциями массива, а также передача массивов в функции (в нашем случае – это функция **`PrintArr)`**.

1.4. **foreach.**

В заключение темы массивов рассмотрим цикл `foreach`, который вы выучили на предыдущем уроке. Как вы помните, его предназначение – поочередный перебор элементов коллекции от начала до конца. Данный цикл удобен тем, что при работе с массивами вам не придется вводить переменные для пробежки по массиву, учитывать его длину и следить за приращением. `foreach` все это делает сам. Единственное, что не позволяет отказаться от циклов `for` и `while` в пользу – это то, что данный цикл работает в режиме чтения, и записать какие-либо данные в элемент массива при «пробежке» будет невозможно.

```
static void Main(string[] args)
{
      int[] myArr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

      foreach (int i in myArr)
      {
            Console.Write(i + " ");
      }
      Console.WriteLine();
}
```

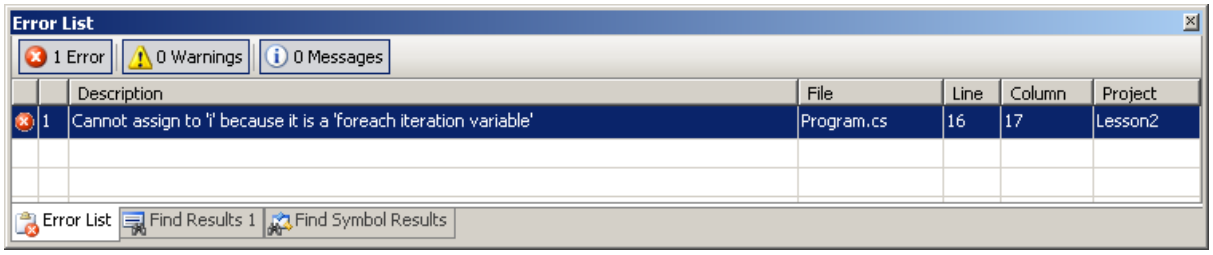
Результат выполнения:



А вот пример, который иллюстрирует, что бывает, если все-таки появится желание записать через `foreach` значение в массив – попробуем обнулить элементы массива:

```
static void Main(string[] args)
{
	int[] myArr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

	foreach (int i in myArr)
	{
		i = 0;
	}
}
```

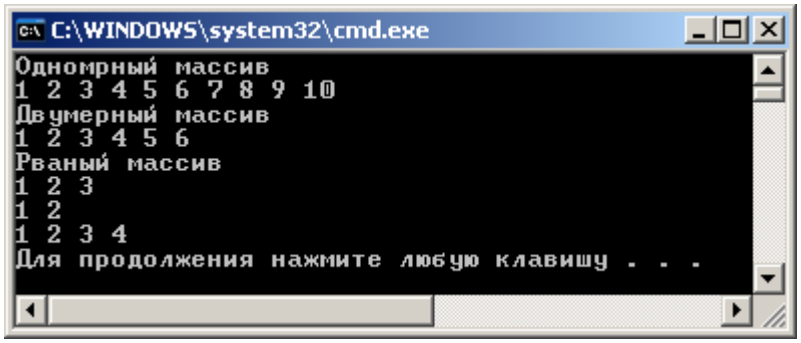
В результате получаем ошибку компиляции:



При работе с многомерными массивами цикл `foreach` не совсем удобен, потому что он выведет элементы всех измерений в одну строку.

```
static void Main(string[] args)
{
      int[] myArr1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
      int[,] myArr2 = { {1, 2, 3}, {4, 5, 6} };
      int[][] myArr3 = new int[3][]{new int[3]{1,2,3},
                                    new int[2]{1,2},
                                    new int[4]{1,2,3,4}};

      Console.WriteLine("Одномрный массив");
      foreach (int i in myArr1)
      {
            Console.Write(i + " ");
      }
      Console.WriteLine("\nДвумерный массив");
      foreach (int i in myArr2)
      {
            Console.Write(i + " ");
      }
      Console.WriteLine("\nРваный массив");
      for (int i = 0; i < myArr3.Length; ++i)
      {
            foreach (int j in myArr3[i])
            {
                  Console.Write(j + " ");
            }
            Console.WriteLine();
      }
}
```

Результаты выполнения:



2. Строки

2.1. Создание строки

Сталкиваясь со строками повсеместно, было бы неправильно думать, что в C# нет удобного способа работы с ними. Именно поэтому и был создан класс `System.String`, от которого происходят все строки. Сам тип `System.String` – ссылочный. Из этого следует то, что строки размещаются в «куче» и имеют богатый

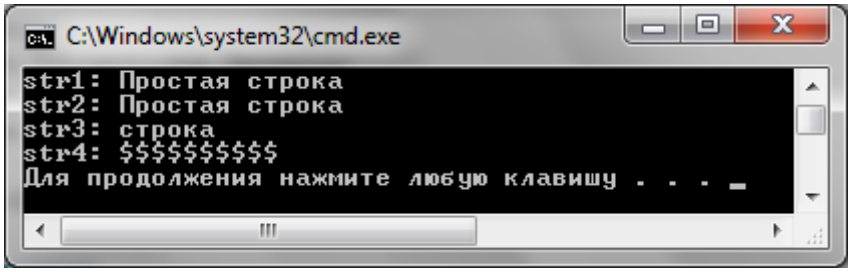
набор методов. Несмотря на то, что строка это ссылка – создавать ее удобно без ключевого слова new.

string <имя> = значение;

Несмотря на такой простой способ создания строки, класс `System.String` имеет 8 конструкторов.

```
string str1 = "Простая строка";
char[] chrArr={'П','р','о','с','т','а','я',' ','с','т','р','о','к','а'};
string str2 = new string(chrArr);
string str3 = new string(chrArr, 8, 6);
string str4 = new string('$', 10);
Console.WriteLine("str1: " + str1);
Console.WriteLine("str2: " + str2);
Console.WriteLine("str3: " + str3);
Console.WriteLine("str4: " + str4);
```

Результат выполнения программы:

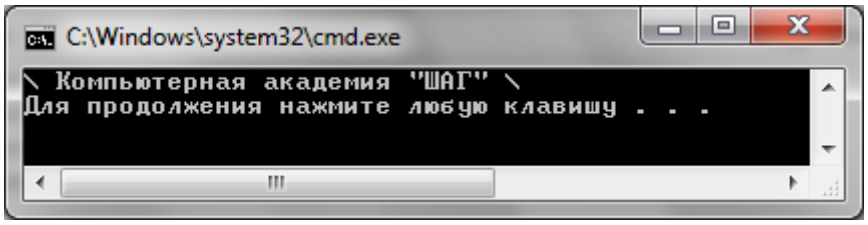


Одно из преимуществ «ссылочности» строки заключается в том, что если нам в программе понадобится множество экземпляров одной и той же строки, то на самом деле память выделится, только лишь для одной. Остальные – будут ссылаться на это место в памяти.

Точно так же как и в языке C++ – в C# существует последовательность управляющих символов. Все символы этой последовательности начинаются с символа «\». Поэтому если мы хотим включить символы обратной косой черты, одинарной или двойной кавычки – непосредственно в строковый литерал, то понадобится указать дополнительный символ «\» перед вышеперечисленными.

```
string str = "\\Компьютерная академия \"ШАГ\"\\";
Console.WriteLine(str);
```

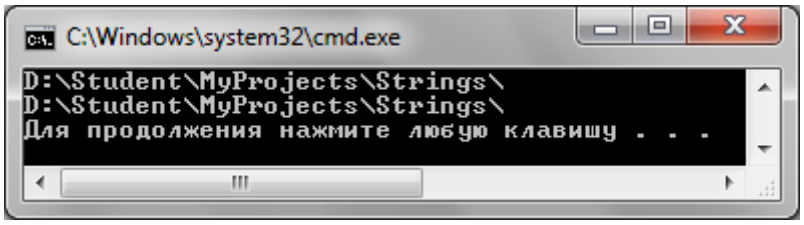
Результат выполнения:



Так как часто возникает необходимость работы с путями к файлам или папкам – было введено понятие «буквальных» строк. Перед такими строками ставится символ «@» и все символы строкового литерала воспринимаются буквально, как они есть.

```
string strPath1 = "D:\\Student\\MyProjects\\Strings\\";
string strPath2 = @"D:\Student\MyProjects\Strings\";
Console.WriteLine(strPath1);
Console.WriteLine(strPath2);
```

В результате получаем два одинаковых вывода:



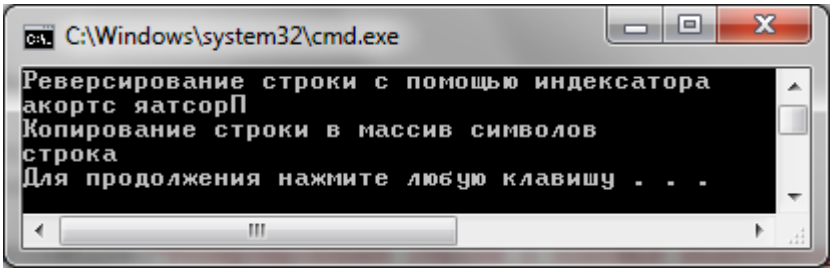
2.2. Операции со строками

Раз уж и существует такой полезный класс как `System.String`, то наверняка в нем предусмотрено достаточное количество методов для комфортной работы, поэтому проведем обзор того, что предоставили нам разработчики для комфортной работы со строками:

- Индексатор – позволяет по индексу получить символ строки. Работает только в режиме чтения.
- Свойство `Length` – возвращает длину строки.
- Метод `CopyTo` – копирует заданное количество символов в массив `char`

```
static void Functions1()
{
      string str = "Простая строка";
      char[] chrArr = new char[6];

      Console.WriteLine("Реверсирование строки с помощью индексатора");
      for (int i = str.Length - 1; i >= 0; --i)
            Console.Write(str[i]);

      Console.WriteLine("\nКопирование строки в массив символов");
      //Копируем шесть символов начиная с восьмой позиции и помещаем в
      //массив
      str.CopyTo(8, chrArr, 0, 6);
      Console.WriteLine(chrArr);
}
```

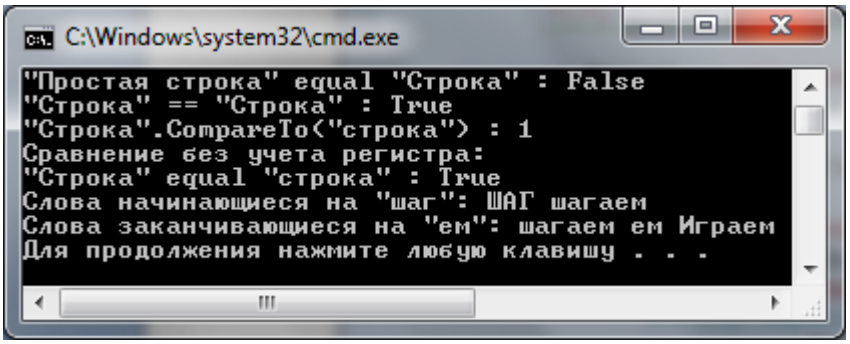
Результат выполнения:



- Методы `Equals`, `Compare`, `CompareTo` и оператор «==» – используются для сравнения строк. Некоторые методы могут принимать параметр типа `StringComparison`, который задает способ сравнения. Например, вариант `CurrentCultureIgnoreCase` используется для сравнения без учета регистра с текущими настройками культуры. Метод `Compare` – статический, поэтому вызывается из под класса. `CompareTo` как и в «сишном» варианте возвращает целое значение(«-1» – левое слово меньше правого, «0» – слова равны, «1» – левое слово больше).
- Методы `StartsWith` и `EndsWith` – проверяют, начинается (заканчивается) ли строка заданным строковым литералом.

```
static void Functions2()
{
      string str1 = "Простая строка";
      string str2 = "Строка";
      string str3 = "строка";
      string[] strArr = { "ШАГ", "шагаем", "бежим", "ем", "Играем" };

      Console.WriteLine("\"" + str1 + "\" equal \"" + str2 + "\" : " +
            str1.Equals(str2));
      Console.WriteLine("\"" + str2 + "\" == \"Строка\" : " +
            (str2 == "Строка"));
      Console.WriteLine("\"" + str2 + "\".CompareTo(\"" + str3 +
            "\") : " + str2.CompareTo(str3));

      Console.WriteLine("Сравнение без учета регистра:");
      Console.WriteLine("\"" + str2 + "\" equal \"" + str3 + "\" : " +
            str2.Equals(str3,
            StringComparison.CurrentCultureIgnoreCase));

      Console.Write("Слова начинающиеся на \"шаг\": ");
      foreach(string s in strArr)
            if(s.StartsWith("шаг",
                  StringComparison.CurrentCultureIgnoreCase))
                    Console.Write(s + " ");
            Console.Write("\nСлова заканчивающиеся на \"ем\": ");
            foreach(string s in strArr)
                if(s.EndsWith("ем",
                  StringComparison.CurrentCultureIgnoreCase))
                  Console.Write(s + " ");
            Console.WriteLine();
}
```

Результат выполнения:

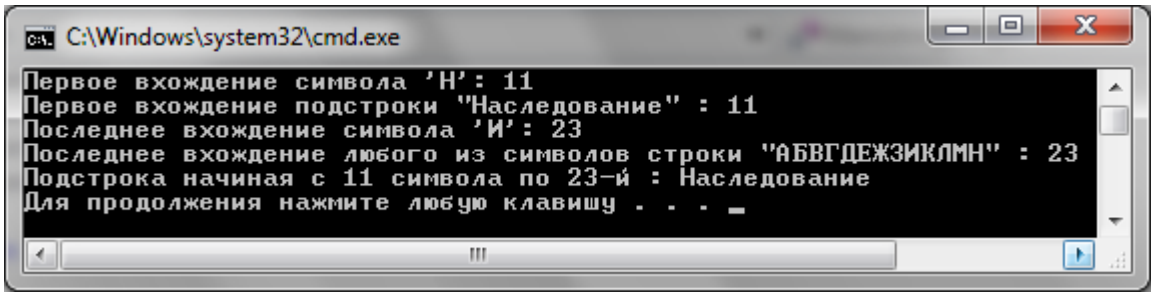


- Метод `IndexOf` и `LastIndexOf` – возвращает индекс первого/последнего вхождения символа/подстроки в исходной строке.
- Методы `IndexOfAny` и `LastIndexOfAny` возвращает индекс первого/последнего вхождения любого из перечисленных символов в исходной строке.
- Медод `SubString` получает подстроку из исходной.

Все методы поиска включают перегруженные версии для поиска в заданном диапазоне с заданным способом сравнения.

```
static void Main(string[] args)
{
      string str1 = "ПолиморфизмНаследованиеИнкапсуляция";
      string str2 = "АБВГДЕЖЗИКЛМН";

      Console.WriteLine("Первое вхождение символа \'Н\': " +
            str1.IndexOf('Н'));
      Console.WriteLine("Первое вхождение подстроки \"Наследование\" : "
            + str1.IndexOf('Н'));
      Console.WriteLine("Последнее вхождение символа \'И\': " +
            str1.LastIndexOf('И'));
      Console.WriteLine("Последнее вхождение любого из символов строки"+
            " АБВГДЕЖЗИКЛМН\ : " +
            str1.LastIndexOfAny(str2.ToCharArray()));
      Console.WriteLine("Подстрока начиная с 11 символа по 23-й : " +
            str1.Substring(11, 12));
}
```

Результат выполнения:



- Метод `Concat` осуществляет конкатенацию (склеивание) строк. Удобная альтернатива данному методу – операции «+» и «+=».
- Методы `ToLower` и `ToUpper` – возвращают строку в нижнем и верхнем регистре соответственно.
- Метод `Replace` заменяет все вхождения символа/подстроки на заданный символ/подстроку.
- Метод `Contains` – проверяет, входит ли заданный символ/подстрока в исходную строку.
- Метод `Insert` – вставляет подстроку в заданную позицию исходной строки.
- Метод `Remove` – удаляет заданный диапазон исходной строки.
- Методы `PadLeft` и `PadRight` дополняют исходную строку заданными символами слева/справа. Если символ не указывается, то дополнение происходит символом пробела. Первый параметр указывает на количество символов в строке, до которого она должна быть дополнена.

- Метод `Split` разрезает строку по заданным символам разделителям. Возвращает массив получившихся в результате нарезания строк. Чтоб исключить из этого массива пробельные строки – нужно использовать данную функцию с параметром `StringSplitOptions.RemoveEmptyEntries`.
- Статический метод `Join` объединяет строки заданного массива в одну и чередует их с указанным символом-разделителем.
- Методы `TrimLeft` и `TrimRight` убирают пробельные (по умолчанию) и заданные символы соответственно с начала и конца строки. Метод `Trim` – делает тоже с обеих сторон строки.
- Статический метод `Format` – позволяет удобно сформатировать строку. Первый параметр – это форматная строка, которая содержит текст выводимый на экран. Если в эту строку необходимо вставить значения переменных, то место вставки помечается индексом в фигурных скобках, при необходимости, там же можно указать количество символов, занимаемых вставляемым элементом и его спецификатор формата. Сами вставляемые данные указываются следующими параметрами метода. Таким образом, синтаксис использования метода `Format` следующий:

```
String.Format("Печатаемый текст {индекс, размер:спецификатор}", данные);
```

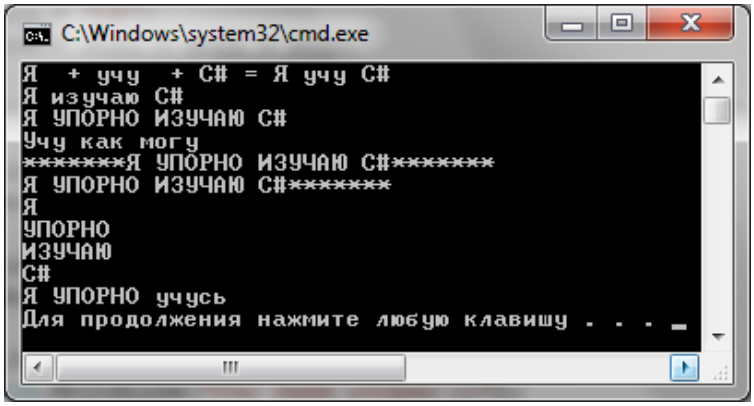
Спецификаторы формата:

1. «`C`» – для числовых данных. Выводит символ местной валюты.
2. «`D`» – для целочисленных данных. Выводит обычное целое число.
3. «`E`» – для числовых данных. Выводит число в экспоненциальной форме.
4. «`F`» – для числовых данных. Выводит число с фиксированной десятичной точкой.
5. «`G`» – для числовых данных. Выводит обычное число.
6. «`N`» – для числовых данных. Выводит числа в формате локальных настроек.
7. «`P`» – для числовых данных. Выводит числа с символом «`%`».
8. «`X`» – для целочисленных данных. Выводит число в шестнадцатеричном формате.

И теперь пример на вышеперечисленные методы:

```
static void Main(string[] args)
{
      string str1 = "Я ";
      string str2 = "учу ";
      string str3 = "C#";
      string str4 = str1 + str2 + str3;

      Console.WriteLine("{0} + {1} + {2} = {3}",str1, str2, str3, str4);

      str4 = str4.Replace("учу", "изучаю");
      Console.WriteLine(str4);

      str4 = str4.Insert(2, "упорно ").ToUpper();
      Console.WriteLine(str4);

      if (str4.Contains("упорно"))
            Console.WriteLine("Учу таки упорно :)");
      else
            Console.WriteLine("Учу как могу");

      str4 = str4.PadLeft(25, '*');
      str4 = str4.PadRight(32, '*');
      Console.WriteLine(str4);
      str4 = str4.TrimStart("*".ToCharArray());
      Console.WriteLine(str4);
      string[] strArr = str4.Split(" *".ToCharArray(),
            StringSplitOptions.RemoveEmptyEntries);
      foreach (string str in strArr)
            Console.WriteLine(str);
      str4 = str4.Remove(9);
      str4 += "учусь";
      Console.WriteLine(str4);
}
```

Результаты выполнения:

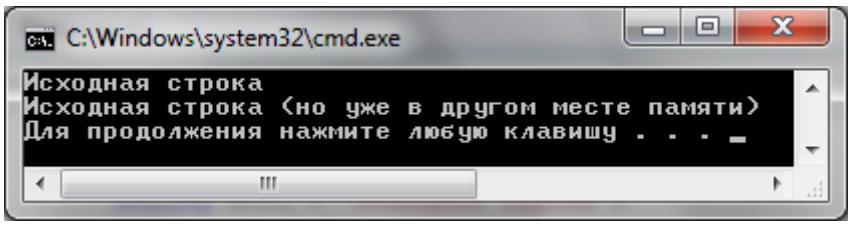


1.1. Особенности использования строк

При работе со строками нужно учитывать тот факт, что в C# строки неизменны. То есть, невозможно внести в строку любые изменения не пересоздав ее. Но беспокоиться по этому поводу не стоит – строка создается и уничтожается автоматически, вам лишь нужно принять ссылку на нее и продолжать работать. При

этом нужно понимать, что ссылочные переменные типа `string` могут менять объекты, на которые они ссылаются. А содержимое созданного `string`-объекта изменить уже невозможно.

```
static void Main(string[] args)
{
      string str1 = "Исходная строка";
      Console.WriteLine(str1);
      str1 += " (но уже в другом месте памяти)";
      Console.WriteLine(str1);
}
```



В приведенном выше примере сначала создается строка на 15 символов. Но при попытке «приклеить» к ней другую строку – выделяется новая память под строку в 46 символов, в которую копируется содержимое обоих исходных. Старая строка при этом попадает под работу «сборщика мусора».

Работа со строками в таком стиле может быть как удобна, так и не очень. К удобствам можно отнести то, что исходная строка не меняется и при случае мы можем к ней обратиться снова. К недостаткам – то, что при интенсивном изменении строки затрачивается много ресурсов (как памяти, так и «сборщика мусора» из-за чего может падать быстродействие).

Для того, чтоб избежать потерь производительности, был придуман класс `StringBuilder`. Данный класс имеет менее обширный набор методов по сравнению с классом `String`, но при этом, работая со строкой данного типа – мы работаем с объектом, расположенным в одном и том же месте в памяти. Память перераспределяется только тогда, когда в строке типа `StringBuilder` не хватает места для произведенных изменений. При этом размер такой строки увеличивается вдвое.

Методы класса `StringBuilder` похожи на «стринговские» – поэтому остановимся на них кратко.

- Метод `Append` – добавляет к исходной строке данные любого из стандартных типов.
- Метод `AppendFormat` – добавляет к исходной строке строку, сформированную в соответствии со спецификаторами формата.
- Метод `Insert` – вставляет данные любого из стандартных типов в исходную строку.
- Метод `Remove` – удаляет из исходной строки диапазон символов.
- Метод `Replace` – заменяет символ/подстроку в исходной строке на указанный символ/подстроку.
- Метод `CopyTo` – копирует символы исходной строки в массив char
- Метод `ToString` – преобразовывает объект `StringBuilder` в String.

Так же в классе `StringBuilder` есть следующие свойства:

- Свойство `Length` – возвращает количество символов, находящихся в строке в данный момент.
- Свойство `Capacity` – возвращает или устанавливает количество символов, которое может быть помещено в строку без дополнительного выделения памяти.
- Свойство `MaxCapacity` возвращает максимальную вместимость строки.

Видим, что класс `StringBuilder` до класса `String` явно «не дотягивает» по удобству использования, но он имеет большое преимущество при интенсивной работе со строками. Поэтому рекомендуется его применение в тех ситуациях, когда производится множественные модификации строки (в результате чего получаем множество строк, ожидающих «сбора мусора»), что приводит к нерациональному расходу памяти. В остальных случаях более удобным является класс `String`.

3. Использование аргументов командной строки

Как и в других языках программирования, в C# программу могут передаваться параметры командной строки. Несколько примеров таких программ - это команда `ping` (в качестве параметра выступает `ip`-адрес компьютера, с которым вы хотите проверить связь), команда `copy` (в качестве параметра выступают имя копируемого файла и новое местоположение и имя).

Для работы с параметрами командной строки существует единственный параметр в функции `main – args` типа `string[]`. Этот параметр содержит передаваемые в программу.

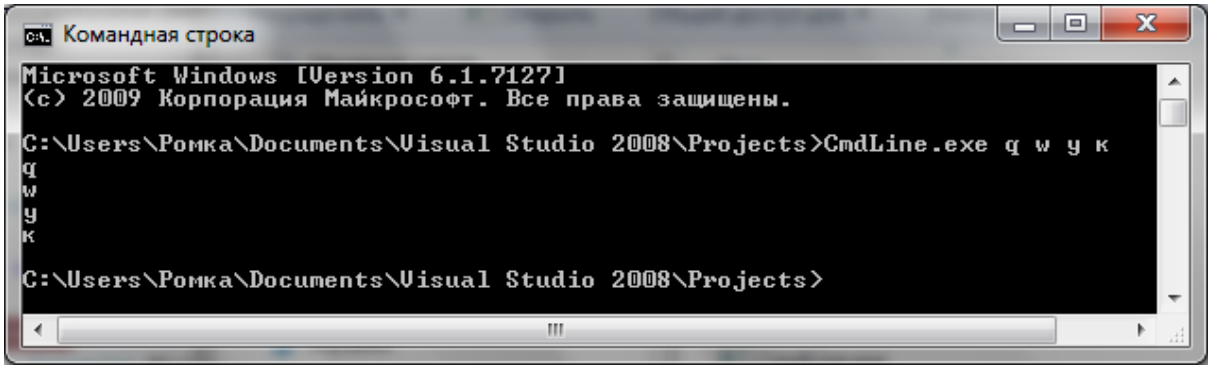
В C++ нулевым параметром командной строки передавалось полное имя исполняемого модуля (т. е. имя файла), в C# нулевой параметр передает первый параметр командной строки. Другими словами, массив `args` содержит только параметры командной строки.

Для получения полного пути исполняемого файла консольной программы предназначен объект `System.Environment.CommandLine`.

Рассмотрим пример функции `main`, которая выведет все свои параметры командной строки:

```
static void Main(string[] args)
{
      foreach(string str in args)
            Console.WriteLine(str);
}
```

Результат выполнения:



Для того, чтоб с командной строкой было удобней работать во время отладки – нужные параметры можно ввести во время проектирования (чтоб запускать программу через IDE). Для этого нужно зайти в свойства проекта и выбрав вкладку «`Debug`»

ввести необходимые параметры в окно «Command line arguments», что проиллюстрировано на рисунке 3.1.

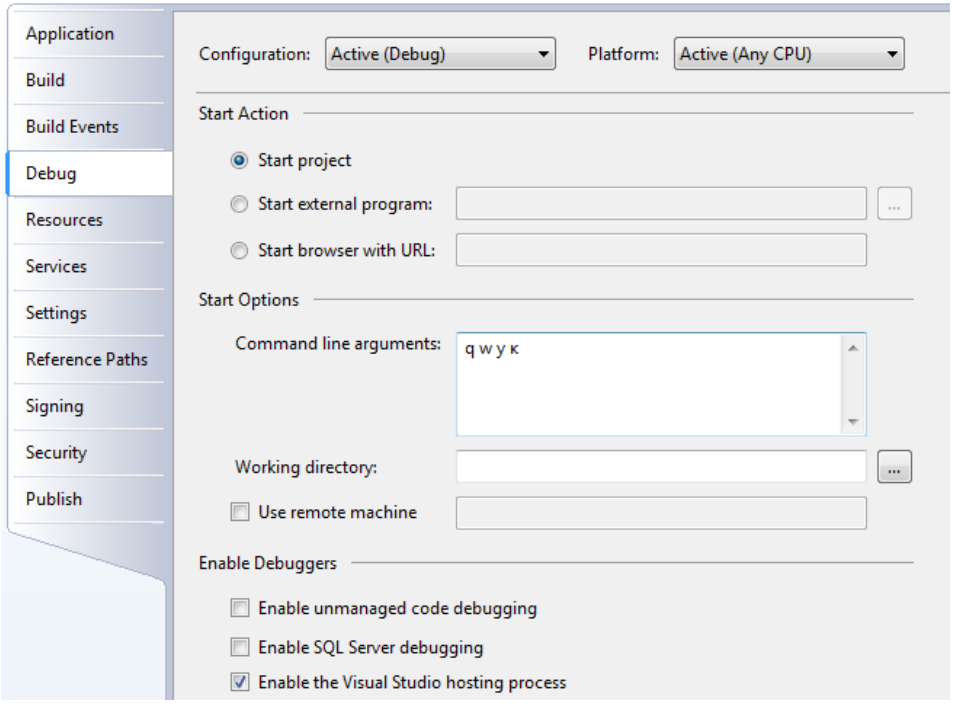


Рис. 1.2. Внесение параметров командной строки через IDE

4. Синтаксис объявления класса

Классы C#, как и в C++ – это некие шаблоны, по которым вы можете создавать объекты. Каждый объект содержит данные и методы, манипулирующие этими данными. Класс определяет, какие данные и какую функциональность может иметь каждый конкретный объект (иногда называемый экземплярам) этого класса. Например, если у вас есть класс, представляющий студента, он может определять такие поля, как `studentID`, `firstName`, `lastName`, `group`, и т.д. которые нужны ему для хранения информации о конкретном студенте.

Класс также может определять функциональность, которая работает с данными, хранящимися в этих полях. Вы создаете экземпляр этого класса для представления конкретного студента, устанавливаете значения полей экземпляра и используете его функциональность. При создании классов, как и всех ссылочных типов – используется ключевое слово new для выделения памяти под экземпляр. В результате объект создастся и инициализируется (помним, что числовые поля инициализируются нулями, логические – `false`, ссылочные – в `null`).

Синтаксис объявления и инициализации класса:

[спецификатор][модификатор] Class <имя класса>

```
{
        [спецификатор][модификатор]   тип <имя поля1>;
        [спецификатор][модификатор]   тип <имя поля2>;
        …
        [спецификатор][модификатор]   тип <Метод1()>
        {
                ...
        }
        [спецификатор][модификатор]   тип <Метод2()>
        {
                ...
        }

}
```

Пример объявления класса описывающего студента:

```
class Student
{
      public int studentID;
      public string firstName;
      public string lastName;
      public string group;
}

class Program
{
      static void Main(string[] args)
      {
            Student st1;
            st1 = new Student();
            Student st2 = new Student();
      }
}
```

Доступ к полям и методам класса, как и в «C++», осуществляется через «.» из-под объекта класса. Также следует напомнить, что доступ к содержимому класса вне его границ, мы имеем только к общедоступным данным. Остальные остаются инкапсулированными. Со спецификаторами и модификаторами мы познакомимся далее.

5. Спецификаторы доступа языка программирования C#

С помощью спецификаторов доступа мы можем регулировать доступность некоторого типа или данных внутри типа.

При определении класса с видимостью в рамках файла, а не другого класса, его можно сделать открытым (`public`) или внутренним (`internal`). Открытый тип доступен любому коду любой сборки. Внутренний класс доступен только из сборки, где он определен. По умолчанию компилятор C# делает класс внутренним.

При определении члена класса (в том числе вложенного) можно указать спецификатор доступа к члену. Спецификаторы определяют, на какие члены можно ссылаться из кода. В CLR определен свой набор возможных спецификаторов доступа, но в каждом языке программирования существует свой синтаксис и термины. Рассмотрим спецификаторы определяющие уровень ограничения – от максимального (`private`) до минимального (`public`):

- `private` – данные доступны только методам внутри класса и вложенных в него классам
- `protected` – данные доступны только методам внутри класса (и вложенным в него классам) или одном из его производных классов
- `internal` – данные доступны только методам в сборке
- `protected internal` – данные доступны только методам вложенного или производного типа класса и любым методам сборки
- `public` – данные доступны всем методам во всех сборках

Вы также должны понимать, что доступ к члену класса можно получить, только если он определен в видимом классе. То есть, если в сборке А определен внутренний класс, имеющий открытый метод, то код сборки Б не сможет вызвать открытый метод, поскольку внутренний класс сборки А не доступен из Б.

В процессе компиляции кода компилятор проверяет корректность обращения кода к классам и членам. Обнаружив некорректную ссылку на какие-либо классы или члены, выдается ошибка компиляции.

Если не указать явно спецификатор доступа, компилятор C# выберет по умолчанию закрытый – наиболее строгий из всех.

Если в производном классе переопределяется член базового – компилятор C# потребует, чтобы у членов базового и производного классов был одинаковый спецификатор доступа. При наследовании базовому классу CLR позволяет понижать, но не повышать уровень доступа к члену.

6. Поля класса

Любой класс может включать в себя множество данных. К таким данным относятся: константы, поля, конструкторы экземпляров класса, конструкторы класса, методы, перегруженные операторы, свойства, события, а также вложенные классы. Остановимся для начала на полях.

Поле – это переменные, которые хранит значение любого стандартного типа или ссылку на ссылочный тип. При объявлении полей могут указываться следующие модификаторы:

- Если модификатор не указывать, то это означает, что поле связано с экземпляром класса, а не самим классом.
- Модификатор `static` – означает, что поле является частью класса, а не объекта.
- Модификатор `readonly` – означает, что поле будет использоваться только для чтения и запись в поле разрешается только из кода метода конструктора либо сразу при объявлении (не путать с константами).

CLR поддерживает изменяемые (read/write) и неизменяемые (`readonly`) поля. Большинство полей – изменяемые. Это означает, что значение таких полей может многократно меняться во время исполнения кода. Такие поля вы уже видели в предыдущем примере в классе `Student`. Неизменяемые поля вроде бы сродни константам, но являются более гибкими, так как значение константам можно задать только при объявлении, а у неизменяемых полей это еще можно сделать и в конструкторах. Важно понимать, что неизменность поля ссылочного типа означает неизменность ссылки, которую оно содержит, но только не объекта, на которую эта ссылка указывает. То есть перенаправить ссылку на другое место в памяти мы не можем, но изменить значение объекта, на который указывает ссылка – можем. Значения неизменяемых полей значимых типов – изменять не можем.

Рассмотрим пример:

```
class MyClass
{
	public readonly int var1 = 10;
	public readonly int[] myArr = { 1, 2, 3 };
	public readonly int var2;                    // инициализация readonly
                                                 // поля при объявлении
	public MyClass(int i)
	{
		var2 = i;                                // инициализация readonly
	}                                            // поля в конструкторе
}

class Program
{
	static void Main(string[] args)
	{
		MyClass obj = new MyClass();
		obj.var = 100;                           // Ошибка
		obj.myArr = new int[10];                 // Ошибка
		obj.myArr[0] = 11;                       //Ошибки нет
	}
}
```

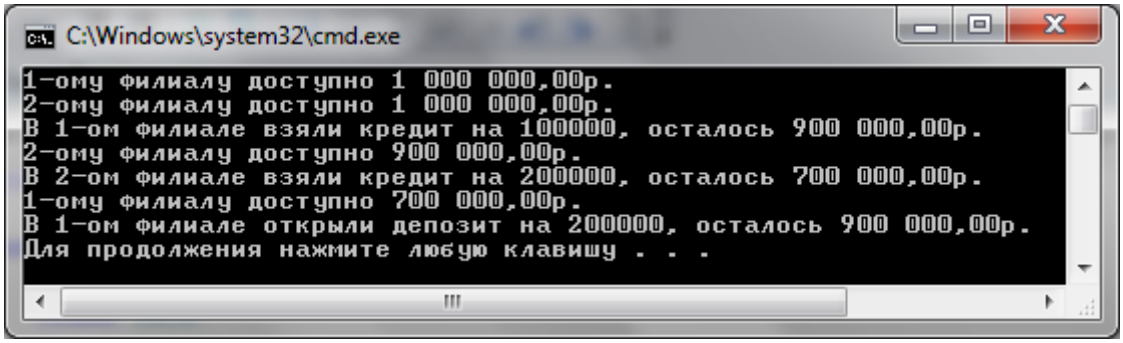
Если объявляется статическое поле, то оно принадлежит классу в целом, а не конкретному объекту. И соответственно получить доступ к такому объекту можно только из-под класса, используя следующий синтаксис:

<Имя класса>.<имя поля>

Например, пускай у нас есть класс `Bank`. В этом классе будет статическое поле `balance`. Сымитируем ситуацию, когда в любом филиале банка можно будет положить деньги на депозит или взять кредит. Пусть все филиалы работают с общим счетом.

```
class Bank
{
      public static float balance = 1000000;
}
class Program
{
      static void Main(string[] args)
      {
            Bank filial1 = new Bank();
            Bank filial2 = new Bank();

            Console.WriteLine("1-ому филиалу доступно {0:C}",
                  Bank.balance);
            Console.WriteLine("2-ому филиалу доступно {0:C}",
                  Bank.balance);
            Console.WriteLine("В 1-ом филиале взяли кредит на 100000" +
                  ", осталось {0:C}", Bank.balance-=100000);
            Console.WriteLine("2-ому филиалу доступно {0:C}",
                  Bank.balance);
            Console.WriteLine("В 2-ом филиале взяли кредит на 200000" +
                  ", осталось {0:C}", Bank.balance -= 200000);
            Console.WriteLine("1-ому филиалу доступно {0:C}",
                  Bank.balance);
            Console.WriteLine("В 1-ом филиале открыли депозит на " +
                  "200000, осталось {0:C}", Bank.balance += 200000);
        }
}
```

Результаты выполнения:



7. Методы класса

В языках C и C++ можно было определить глобальные функции, которые были не связаны с определенным классом. В C# этого сделать не получится. То есть все функции обязательно должны определяться внутри классов или структур.

Синтаксис объявления методов похож на объявления методов в C-подобных языках, и практически идентичен синтаксису C++. Основное отличие от C++ заключается в том, что в C# каждый метод отдельно объявлен как общедоступный или приватный. То есть блоки `public:`, `private:` для группировки нескольких методов использовать нельзя.

Также все методы C# объявляются и определяются внутри определения класса, таким образом нельзя отделить реализацию метода от объявления, как это делается в C++.

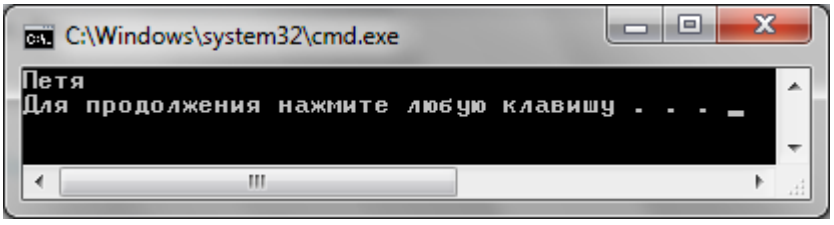
В C# определение метода состоит из спецификаторов и модификаторов, типа возвращаемого значения, за которым следует имя метода, затем – список аргументов (если они есть), заключенный в скобки, и далее – тело метода, заключенное в фигурные скобки:

```
[спецификаторы][модификаторы] тип_возврата <Имя Метода>([параметры])
{
      // Тело метода
}
```

Добавим в класс `Student` метод. При этом сделаем все поля класса закрытыми. Используйте практику сокрытия данных класса и предоставления открытых методов по работе с этими данными. Тем самым вы поддерживаете принцип инкапсуляции данных и уменьшаете вероятность порчи информации в них.

```
class Student
{
      private string firstName = "Петя";

      public void ShowName()
      {
            Console.WriteLine(firstName);
      }
}

class Program
{
      static void Main(string[] args)
      {
            Student st = new Student();
            st.ShowName();
      }
}
```

Результат выполнения:



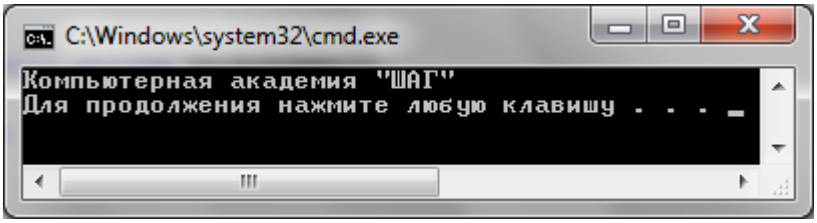
Специально для работы со статическими полями – были введены статические методы. Эти методы, как и статические поля, принадлежат классу, а не объекту. Они

исключают возможность вызова из-под объекта и соответственно не работают с нестатическими полями.

Предположим, что все создаваемые нами студенты будут студентами академии «ШАГ» и соответственно добавим в наш класс `Student` статическое поле, которое будет содержать имя учебного заведения. Чтоб поле нельзя было изменить – сделаем его закрытым и позволим статическому методу `ShowAcademy` работать с нашим полем в режиме чтения.

```
class Student
{
	private static string academyName="Компьютерная академия \"ШАГ\"";

	// старые поля и методы остаются без изменения

	public static void ShowAcademy ()
	{
		Console.WriteLine(academyName);
	}

}

class Program
{
	static void Main(string[] args)
	{
		Student.ShowAcademy();
	}
}
```

Результат выполнения:



7.1. Передача параметров

Аргументы могут передаваться методу либо по значению, либо по ссылке. Когда переменная передастся по ссылке, вызываемый метод получает саму переменную, поэтому любые изменения, которым производятся над переменной внутри метода, останутся в силе после его завершения. С другой стороны, если переменная передается по значению, то вызываемый метод получает копию этой переменной, и соответственно все изменения в копии по завершении метода будут

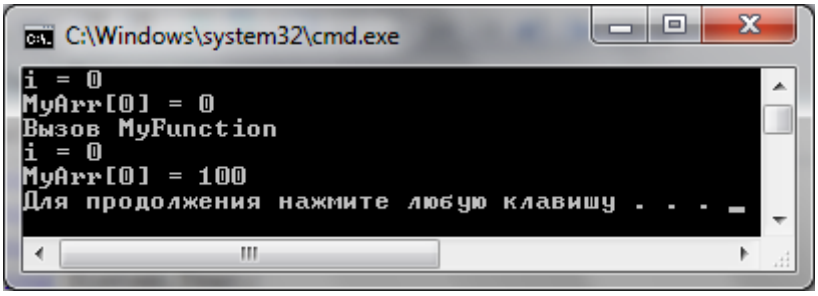
утеряны. Для сложных типов данных передача по ссылке более эффективна из-за большого объема данных, который приходится копировать при передаче по значению.

В C# все параметры передаются по значению, если вы не укажете обратное. Это поведение аналогично C++. Однако нужно быть осторожным с пониманием этого в отношении ссылочных типов. Поскольку переменная ссылочного тина содержит лишь ссылку на объект, то именно ссылка будет скопирована при передаче параметра, а не сам объект. То есть изменения, произведенные в самом объекте, сохранятся. В отличие от этого переменные типа значений действительно содержат данные, поэтому в метод передается копия самих данных. Например, `int` передастся в метод по значению, и любые изменения, которые сделает метод в этом `int`. не изменят значения исходного объекта. В противоположность этому, если в метод передастся массив или любой другой ссылочный тип, такой как класс, и метод использует эту ссылку для изменения значения в массиве, то это повое значение будет отражено в исходном объекте массива.

Рассмотрим пример вышесказанного:

```
class Program
{
      static void MyFunctionByVal1(int[] myArr, int i)
      {
            myArr[0] = 100;
            i = 100;
      }
      static void Main(string[] args)
      {
            int i = 0;
            int[] myArr = { 0, 1, 2, 4 };
            Console.WriteLine("i = {0}", i);
            Console.WriteLine("MyArr[0] = {0}", myArr[0]);
            Console.WriteLine("Вызов MyFunction");
            MyFunctionByVal1 (myArr, i);
            Console.WriteLine("i = {0}", i);
            Console.WriteLine("MyArr[0] = {0}", myArr[0]);
      }
}
```

Результаты выполнения:



7.2. Ключевое слово return

В выше приведенных примерах методы не возвращали никакой информации. Но так же как и в C++ в C# методы могут возвращать данные. Синтаксис такого метода абсолютно идентичный «сишному»: в заголовке метода добавляется тип возвращаемого значения, а само значение возвращается из метода с помощью ключевого слова `return`.

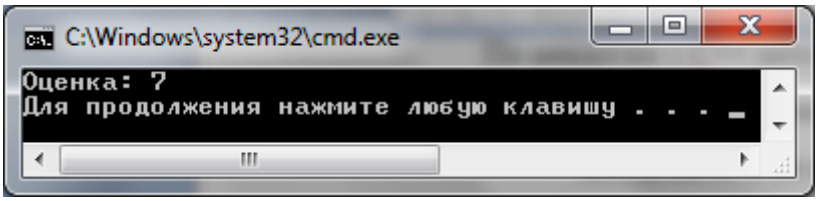
По аналогии с C++ метод может завершить свое выполнение тремя способами:

1. Когда управление дойдет до завершающей фигурной скобки (при этом метод ничего не возвращает)
2. Когда управление дойдет до ключевого слова return (без возвращаемого значения и тип возвращаемого значения метода – `void`)
3. Когда управление дойдет до ключевого слова return (после которого стоит возвращаемое значение, метод что-либо возвращает)

Добавим нашему студенту метод, который будет возвращать заработанную им оценку. Сама оценка будет генерироваться случайно (для чего будем использовать класс `Random`).

```
class Student
{
	// старые поля и методы остаются без изменения

	public int GetMark()
	{
		return new Random().Next(1, 12);
	}

}

class Program
{
	static void Main(string[] args)
	{
		Student st = new Student();
		Console.WriteLine("Оценка: {0}", st.GetMark());
	}
}
```

Результат выполнения:



7.3. Перегрузка методов

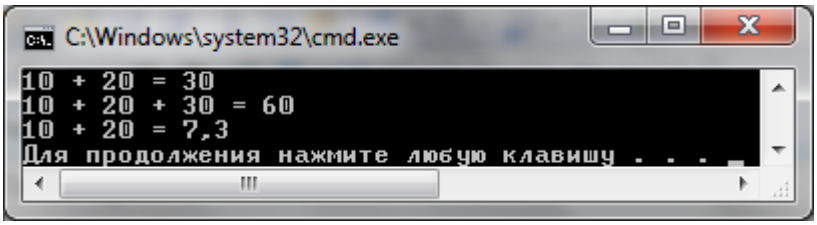
Перегрузка методов – определение нескольких методов с одинаковым именем и разной сигнатурой.

Еще раз уточним, что методы должны отличаться только сигнатурой (количеством параметром, типами параметров или порядком следования параметров). Тип возвращаемого методом значения, так же как и модификаторы на перегрузку не влияет!

Рассмотрим пример, где создается класс `Mathematic`, в котором будет несколько перегруженных методов по сложению чисел.

```
class Mathematic
{
	public static int Sum(int a, int b)
	{
		return a + b;
	}
	public static int Sum(int a, int b, int c)
	{
		return a + b + c;
	}
	public static double Sum(double a, double b)
	{
		return a + b;
	}
}
class Program
{
	static void Main(string[] args)
	{
		int a = 10, b = 20, c = 30;
		double da = 2.5, db = 4.8;
		Console.WriteLine("{0} + {1} = {2}", a, b, Mathematic.Sum(a,
			b));
		Console.WriteLine("{0} + {1} + {2} = {3}", a, b, c,
			Mathematic.Sum(a, b, c));
		Console.WriteLine("{0} + {1} = {2}", a, b,
			Mathematic.Sum(da, db));
	}
}
```

Результаты выполнения:



8. Конструкторы

8.1. Понятие конструктора

Конструкторы – это методы класса, которые вызываются при создании объекта.

В C# конструкторов существует 3 вида. Несмотря на такое разнообразие, идея каждого – выполнить некоторые действия при создании объектов. Рассмотрим их по порядку.

Конструктор по умолчанию – не принимает никаких параметров и предоставляется в «подарок» при создании класса. В отличие от конструктора по умолчанию в C++ (который предоставляется компилятором), этот конструктор полезен тем, что он обнуляет все числовые типы, устанавливает в `false` логический, и в `null` – ссылочный. Итого после создания объекта вы ни в одном его поле не увидите мусора. В случае необходимости вы можете переопределить конструктор по умолчанию под ваши нужды.

- Конструктор с параметрами – конструктор, который может принимать необходимое количество параметров для инициализации полей класса или каких-либо других действий.

Статический конструктор – конструктор, относящийся к классу, а не к объекту. Существует для инициализации статических полей класса. Определяется без какого-либо спецификатора доступа с ключевым словом `static`.

При создании конструкторов нужно помнить, что все конструкторы (кроме статического) имеют спецификатор доступа `public`, имя, совпадающее с именем класса, и ничего не возвращают (даже `void`), также все, кроме конструктора по

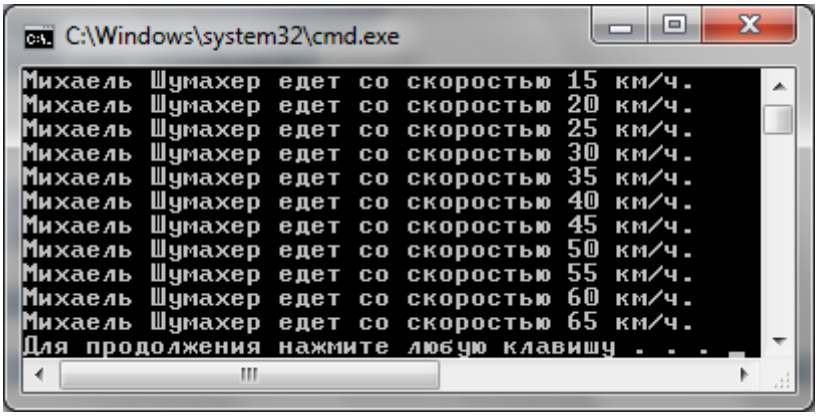
умолчанию и статического конструктора, могут иметь необходимое количество параметров. Конструктор по умолчанию может быть только один.

Еще один важный момент: если вы взялись за определение любого конструктора, то конструктор по умолчанию, который вам предоставлялся компилятором работать не будет!!!

Рассмотрим пример определения своего конструктора по умолчанию, а остальные будут рассмотрены в следующих разделах. Для работы с конструкторами создадим новый класс, описывающий машину.

```
class Car
{
      private string driverName;          // Имя водителя
      private int currSpeed;              // Текущая скорость

      public Car()                        // Конструктор по умолчанию
      {
            driverName = "Михаель Шумахер";
            currSpeed = 10;
      }
      public void PrintState()            // Распечатка текущих данных
      {
            Console.WriteLine("{0} едет со скоростью {1} км/ч.",
                  driverName, currSpeed);
      }
      public void SpeedUp(int delta)      // Увеличение скорости
      {
            currSpeed += delta;
      }
}
class Program
{
      static void Main(string[] args)
      {
            Car myCar = new Car();
            for (int i = 0; i <= 10; i++)
            {
                  myCar.SpeedUp(5);
                  myCar.PrintState();
            }
      }
}
```

Результат выполнения:



8.2. Параметризованный конструктор

Как было сказано выше конструктор с параметрами отличается от конструктора по умолчанию собственно наличием параметров. Поэтому сразу перейдем к примеру и добавим в наш класс Car параметризованный конструктор. Для экономии места старые поля и методы в примере отсутствуют, но в полной версии – есть.

```
class Car
{
      // Старые поля и методы...

      public Car(string name)
      {
            driverName = name;
            currSpeed = 10;
      }
}

class Program
{
      static void Main(string[] args)
      {
            Car myCar = new Car("Рубенс Барикелло");
            for (int i = 0; i <= 10; i++)
            {
                myCar.SpeedUp(5);
                myCar.PrintState();
            }
      }
}
```

8.3. Перегруженные конструкторы

Поскольку конструкторы – это методы, а, как мы уже знаем, методы можно перегружать, то следовательно и конструкторов может быть сколько угодно. Единственное ограничение – конструктор по умолчанию должен быть один. Ну а количество параметризованных – ограничено только лишь здравым смыслом. Вообще старайтесь определять те конструкторы, с помощью которых было бы удобней создавать объект. Вернемся к классу машины и определим еще пару конструкторов.

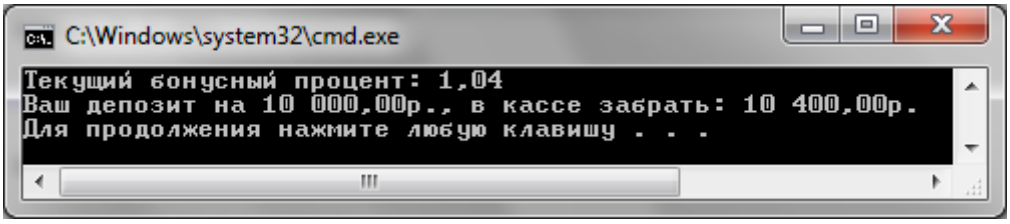
```
class Car
{
      // Старые поля и методы...

      public Car(string name)
      {
            driverName = name;
            currSpeed = 10;
      }
      public Car(string name, int speed)
      {
            driverName = name;
            currSpeed = speed;
      }
}
class Program
{
      static void Main(string[] args)
      {
            Car myCar = new Car("Ральф Шумахер", 10);
            for (int i = 0; i <= 10; i++)
            {
                myCar.SpeedUp(5);
                myCar.PrintState();
            }
      }
}
```

8.4. Статические конструкторы

Статический конструктор связан с классом, а не с конкретным объектом. Сам конструктор нужен для инициализации статических данных. Когда вызывается этот конструктор – неизвестно, но гарантируется, что вызов произойдет до первого создания объекта класса. Для примера со статическим конструктором создадим класс описывающий банковские филиалы, но на этот раз статическое поле будет содержать бонус в процентах для оформления депозитов. А текущий баланс у каждого филиала будет свой.

```
class Bank
{
	private double currBalance;
	private static double bonus;

	public Bank(double balance)
	{
		currBalance = balance;
	}
	static Bank()
	{
		bonus = 1.04;
	}
	public static void SetBonus(double newRate)
	{
		bonus = newRate;
	}
	public static double GetBonus()
	{
		return bonus;
	}
	public double GetPercents(double summa)
	{
		if ((currBalance - summa) > 0)
		{
			double percent = summa * bonus;
			currBalance -= percent;
			return percent;
		}
		return -1;
	}
}
class Program
{
	static void Main(string[] args)
	{
		Bank b1 = new Bank(1000000);
		Console.WriteLine("Текущий бонусный процент: " +
			Bank.GetBonus());
		Console.WriteLine("Ваш депозит на {0:C}, в кассе забрать:"+
			"{1:C}", 10000, b1.GetPercents(10000));
	}
}
```

Результат выполнения:



9. Ключевое слово this

Ключевое слово `this` – это неявно присутствующая в классе ссылка на объект, вызывающий метод (т.е. это ссылка на «самого себя»). Используется в нескольких случаях: если имя параметра метода совпадает с именем поля класса, то при инициализации поля указывается, что поле достается из-под ссылки this; другое применение `this` – это вызов конструкторов. Как правило в классе определяется главный конструктор (как правило главным выбирают конструктор с максимальным количеством параметров), который содержит код инициализации, а все остальные – вызывают главный передавая ему необходимые параметры. Рассмотрим снова пример с классом `Car`, внеся в него следующие изменения:

```
class Car
{
	private string driverName;
	private int currSpeed;

	public Car():this("Нет водителя", 0){}
	public Car(string driverName):this(driverName, 0){}
	public Car(string driverName, int speed) // Главный конструктор
	{
		driverName = driverName;
		currSpeed = 10;
	}
	public void SetDriver(string driverName)
	{
		this.driverName = driverName;
	}
	// Старые методы остались как прежде
}
class Program
{
	static void Main(string[] args)
	{
		Car myCar = new Car();
		for (int i = 0; i <= 10; i++)
		{
			myCar.SpeedUp(5);
			myCar.PrintState();
		}
	}
}
```

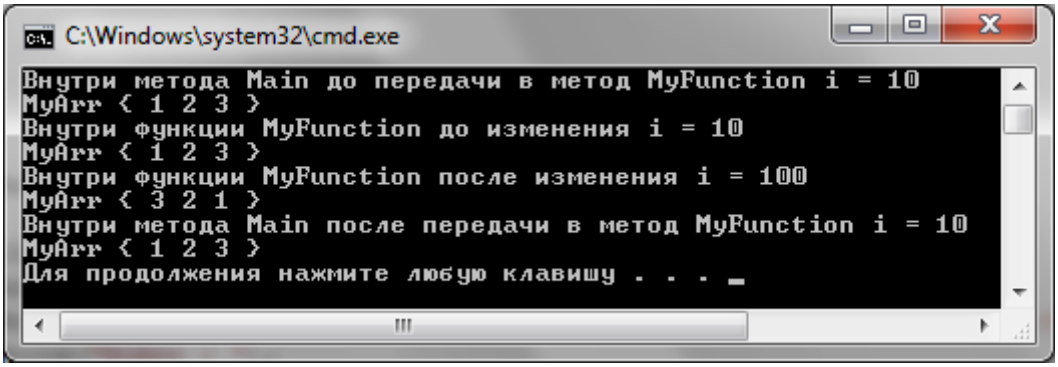
10. Использование ref и out параметров

Как уже было сказано все, что передается в методы – передается по значению (даже, как ни удивительно, ссылочные типы). Для того, чтоб добиться передачи

параметров через ссылку существуют ключевые слова ref и out. Убедимся на примере, в передаче по значению.

```
private static void MyFunctionByVal2(int i, int[] myArr)
{
      Console.WriteLine("Внутри функции MyFunction до изменения i = "+
            "{0}", i);
      Console.Write("MyArr { ");
      foreach (int val in myArr)
            Console.Write(val + " ");
      Console.WriteLine("}");
      i = 100;
      myArr = new int[] { 3, 2, 1 };
      Console.WriteLine("Внутри функции MyFunction после изменения i ="+
            "{0}", i);
      Console.Write("MyArr { ");
      foreach (int val in myArr)
            Console.Write(val + " ");
      Console.WriteLine("}");
}

static void Main(string[] args)
{
      int i = 10;
      int[] myArr = { 1, 2, 3 };
      Console.WriteLine("Внутри метода Main до передачи в метод " +
            "MyFunction i = {0}", i);
      Console.Write("MyArr { ");
      foreach (int val in myArr)
            Console.Write(val + " ");
      Console.WriteLine("}");
      MyFunctionByVal2 (i, myArr);
      Console.WriteLine("Внутри метода Main после передачи в метод "+
            "MyFunction i = {0}", i);
      Console.Write("MyArr { ");
      foreach (int val in myArr)
            Console.Write(val + " ");
      Console.WriteLine("}");
}
```

По результатам выполнения видим, что даже передача массива происходит по значению и в методе `Main` исходный массив не поменялся.

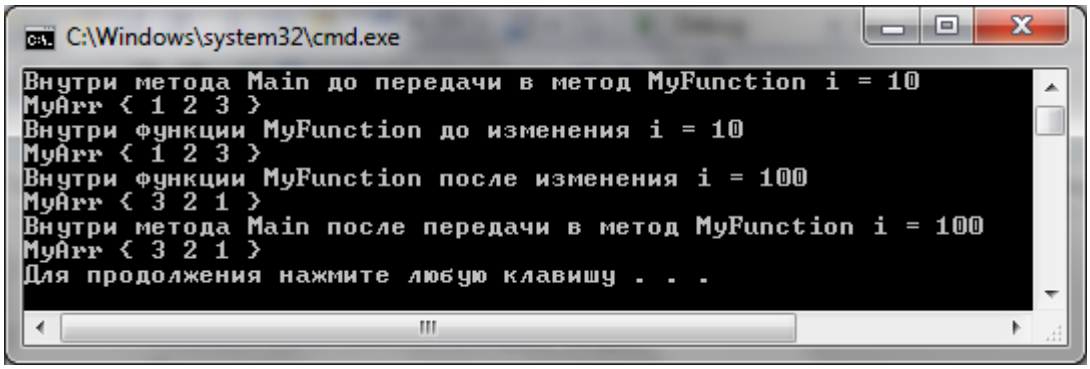


10.1. Использование модификатора ref

Ключевым словом `ref` помечаются те параметры, которые должны передаваться в метод по ссылке. Таким образом, мы будем внутри метода манипулировать данными, объявленными в вызывающем методе. Аргументы, которые передаются в метод с ключевым словом `ref`, обязательно должны быть проинициализированы, иначе компилятор выдаст сообщение об ошибке. Попробуем обозначить параметры из предыдущего примера как ссылочные и посмотрим, как изменится ситуация. Обратите внимание, что даже при вызове метода необходимо еще раз указать, что аргументы передаются по ссылке.

```
private static void MyFunctionByRef(ref int i, ref int[] myArr)
{
      Console.WriteLine("Внутри функции MyFunction до изменения i = "+
            {0}", i);
      Console.Write("MyArr { ");
      foreach (int val in myArr)
            Console.Write(val + " ");
      Console.WriteLine("}");
      i = 100;
      myArr = new int[] { 3, 2, 1 };
      Console.WriteLine("Внутри функции MyFunction после изменения i ="+
            {0}", i);
      Console.Write("MyArr { ");
      foreach (int val in myArr)
            Console.Write(val + " ");
      Console.WriteLine("}");
}

static void Main(string[] args)
{
      int i = 10;
      int[] myArr = { 1, 2, 3 };
      Console.WriteLine("Внутри метода Main до передачи в метод " +
            "MyFunction i = {0}", i);
      Console.Write("MyArr { ");
      foreach (int val in myArr)
            Console.Write(val + " ");
      Console.WriteLine("}");
      MyFunctionByRef(ref i, ref myArr);
      Console.WriteLine("Внутри метода Main после передачи в метод "+
            "MyFunction i = {0}", i);
      Console.Write("MyArr { ");
      foreach (int val in myArr)
            Console.Write(val + " ");
      Console.WriteLine("}");
}
```

Результаты выполнения поменялись:

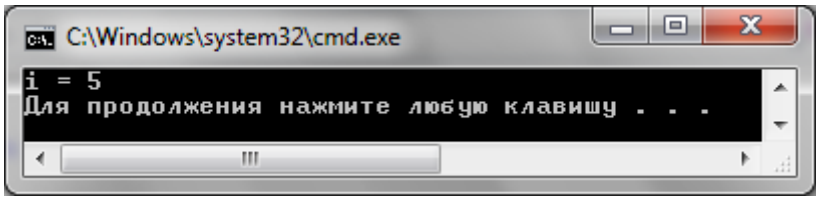


10.2. Использование модификатора out

Параметры, обозначенные ключевым словом `out`, также используются для передачи по ссылке. Отличие от ref состоит в том, что параметр считается выходным и соответственно компилятор разрешит не инициализировать его до передачи в метод и проследит, чтоб метод занес в этот параметр значение (иначе будет выдано сообщение об ошибке).

```
class Program
{
	static void Main(string[] args)
	{
		int i;
		GetDigit(out i);
		Console.WriteLine("i = " + i);
	}
	private static void GetDigit(out int digit)
	{
		digit = new Random().Next(10);
	}
}
```

Результат выполнения:

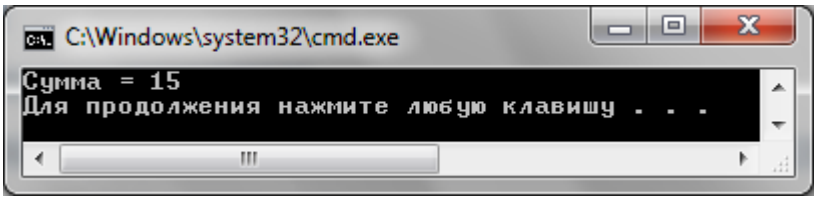


11. Создание методов с переменным количеством аргументов

Для создания метода с переменным количеством аргументов можно было бы в качестве параметра передать массив значений. Рассмотрим пример такого метода.

```
class Program
{
      static void Main(string[] args)
      {
            Console.WriteLine("Сумма = " +
                  Sum(new int[] { 1, 2, 3, 4, 5 }));
      }
      private static int Sum(int[] arr)
      {
            int res = 0;
            foreach (int i in arr)
                res += i;
            return res;
      }
}
```

Результат выполнения:



Видим, что хотя результат правильный, но подход явно некрасивый. Поэтому специально для создания методов с переменным количеством аргументов было придумано ключевое слово `params`, которым помечают параметр метода. При использовании этого ключевого слова необходимо учитывать, что параметр, помечаемый ключевым словом `params`:

- должен стоять последним в списке параметров;
- должен указывать на одномерный массив любого типа.

Рассмотрим тот же пример, но с параметром типа `params`.

```
class Program
{
      static void Main(string[] args)
      {
            Console.WriteLine("Сумма = " + Sum( 1, 2, 3, 4, 5 ));
      }
      private static int Sum(params int[] arr)
      {
            int res = 0;
            foreach (int i in arr)
                res += i;
            return res;
      }
}
```

В результате получаем тот же вывод, но код при этом стал более изящным.

12. Частичные типы (partial types)

Частичные типы поддерживаются только компиляторами C# и некоторых других языков, но CLR ничего о них не знает.

Ключевое слово `partial` говорит компилятору C#, что исходный код класса может располагаться в нескольких файлах. Существуют две основные причины, по которым исходный код разбивается на несколько файлов.

- Управление версиями – определение класса может содержать большое количество кода. Если этот класс будут одновременно редактировать несколько программистов, то по окончании работы им придется каким-то образом объединять свои результаты, что весьма неудобно.
- Разделители кода – при создании в Microsoft Visual Studio нового проекта Windows Forms или Web Forms, некоторые файлы с исходным кодом создаются автоматически. Они называются шаблонными. При использовании конструкторов форм Visual Studio в процессе создания и редактирования элементов управления формы Visual Studio автоматически генерирует весь необходимый код и помещает его в отдельные файлы. Это значительно повышает продуктивность работы. Раньше автоматически генерируемый код помешался в тот же файл, где программист писал свой исходный код. Проблема была в том, что при случайном изменении сгенерированного кода конструктор форм переставал корректно работать. Начиная с Visual Studio 2005, при создании нового проекта Visual Studio создает два исходных

файла; один предназначен для программиста, а в другой помещается код, создаваемый редактором форм. Теперь вероятность случайного изменения генерируемого кода существенно меньше.

Ключевое слово `partial` применяется к типам во всех файлах с определением класса. При компиляции компилятор объединяет эти файлы, и готовый класс помещается в результирующий файл сборки с расширением .exe или dll. Как уже говорилось, частичные типы реализуются только компилятором C#; поэтому все файлы с исходным кодом типа необходимо писать на одном языке и компилировать их в единый модуль.

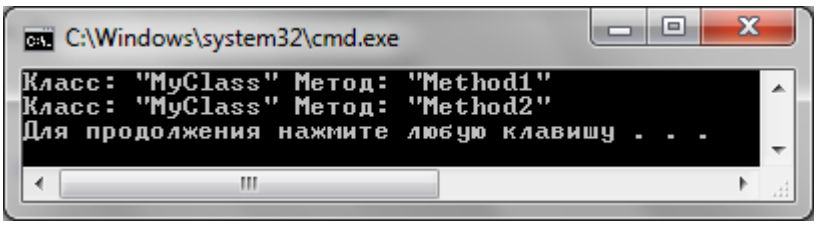
Например:

```
// Первый файл
partial class MyClass
{
	public static void Method1()
	{
		Console.WriteLine("Класс: \"MyClass\" Метод: \"Method1\"");
	}
}
```

```
// Второй файл
partial class MyClass
{
	public static void Method1()
	{
		Console.WriteLine("Класс: \"MyClass\" Метод: \"Method2\"");
	}
}
```

```
// Третий файл

class Program
{
	static void Main(string[] args)
	{
		MyClass.Method1();
		MyClass.Method2();
	}
}
```

Результат получаем так, как будто класс объявлен в одном файле:



На этом данный урок закончен. И как всегда – домашнее задание:

1. Сжать массив, удалив из него все 0 и заполнить освободившиеся справа элементы значениями -1
2. Преобразовать массив так, чтобы сначала шли все отрицательные элементы, а потом положительные(0 считать положительным)
3. Написать программу, которая предлагает пользователю ввести число и считает сколько раз это число встречается в массиве.
4. В двумерном массиве порядка M на N поменяйте местами заданные столбцы.
5. Создать метод принимающий, введенную пользователем, строку и выводящий на экран статистику по этой строке. Статистика должна включать общее количество символов, количество букв (и сколько в верхнем регистре и нижнем), количество цифр, количество символов пунктуации и количество пробельных символов.
6. пользователь вводит строку и символ. В строке найти все вхождения этого символа и перевести его в верхний регистр, а также удалить часть строки, начиная с последнего вхождения этого символа и до конца.
7. Придумать класс описывающий студента и предусмотреть в нем следующие моменты: фамилия, имя, отчество, группа, возраст, массив(рваный) оценок по программированию, администрированию и дизайну. А также добавить методы по работе с перечисленными данными: возможность установки/получения оценки, получение среднего балла по заданному предмету, распечатка данных о студенте.