Microsoft .Net Framework and C# Programming Language

STEP IT ACADEMY

Lesson 2

Arrays and strings. Enumerations

Contents

1. Arrays

One-dimensional arrays

Array is a series of interconnected elements of the same data type.

The program can address any element of the array by specifying a name of an array followed by an integer value enclosed in the square brackets that indicates location of an element within an array, which is called an element index.

According to the Common Language Specification (CLS), numbering of elements in the array should begin with zero. If you follow this rule, then the methods written in C# will be able to pass a reference to an array to the code written in another language, such as Microsoft Visual Basic .NET. In addition, Microsoft did its best to optimize the performance of arrays with initial zero index, because they became widely distributed. However, other options of array indexing are allowed in the CLR, although this is not encouraged.

Here is the syntax for one-dimensional array declaration

```
Array_elements_type [] array_name;
```

Examples of one-dimensional array declaration:

```
int[] myArray;          // Declaring a reference
                        // to an integer array
string [] myStrings;    // Declaring a reference
                        // to a string array
```

In C# all the arrays are inherited from the `System.Array` class. This inheritance means that all the arrays are objects.

On the one hand, it provides many benefits, and on the other hand, it has a range of drawbacks. The benefits include a considerable set of methods for working with arrays inherited from the `System.Array` class, control of going beyond the bounds of an array, etc. The drawbacks include decrease in performance when working with an array due to the fact that it is located in the "heap".

Given that the arrays are of reference type, two empty links were created in the above example. For further work it is necessary to allocate memory for these links; a `new` operator is used for this purpose.

```
myArray = new int[10];        // Allocating memory for
                              // an array of 10 numbers
myStrings = new string[50]; // Allocating memory for
                              // an array of 50 lines
```

After allocating memory, elements are initialized by their default values: values of integer types are set to "*0*", values of real types are set to "*0.0*", values of logical type are set to "*false*", and values of reference types are set to "*null*".

It is also possible to initialize an array with the desired values when declaring:

```
int[] myArray1 = new int[10] {1, 2, 3, 4, 5, 6, 7, 8,
                                 9, 10};
// int[] myArray2 = new int[5]{ 2, 14, 54, 8 }; // Error
int[] myArray3 = new int[] {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int[] myArray4 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

In the first case, if in the initialization the number of elements list is greater or less than ordered, then the compiler generates an error message (Figure 1.1). In the second and third cases, array size is calculated from the number of elements in the initialization list.

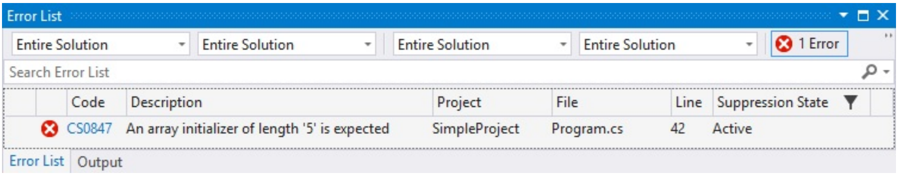


Figure 1.1. Array initialization error

Initialization of an array reference without the use of the new operator will lead to a failure at the compile time (Figure 1.2).

```
int[] myArray;
//myArray = { 2, 14, 54, 8, 11 };    // Error
myArray = new int[] { 2, 14, 54, 8, 11 };    // OK
```

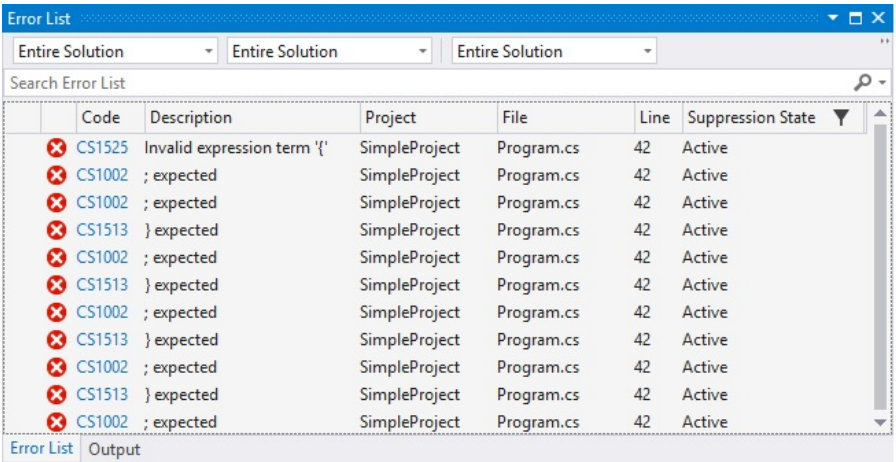


Figure 1.2. Error at the compile time

Addressing the elements of a one-dimensional array is performed by specifying a serial number of an element — index:

```
myArray [2] = 53; //Assigning a value to the third
                  //element of the array
Console.WriteLine(myArray [2]); //Displaying the value
                  //of the third element of the array
```

Since all the arrays are objects, then additional information (size, lower bound, array type, number of measurements, number of elements in each dimension, etc.) is stored in them in addition to value data. Allocation of an array in memory can be schematically represented in the following way:

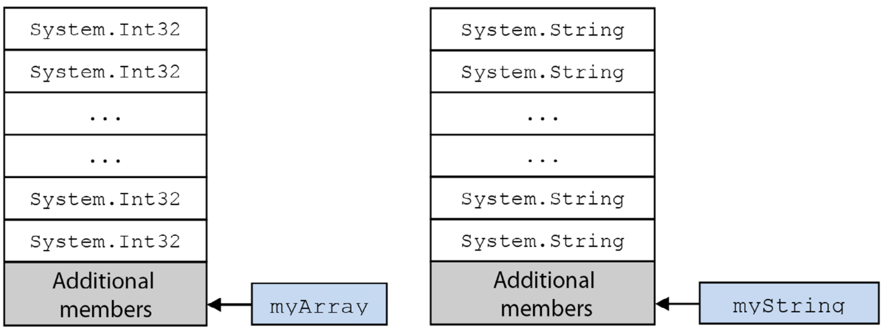


Figure 1.3. Allocation of an array in memory

Multidimensional arrays

Arrays can be not only one-dimensional, but also multi-dimensional, i.e. they can have several dimensions. The syntax of multi-dimensional array is shown below.

```
Array_elements_type [,,…,] array_name;
```

Here are some examples of declaration and initialization of two-dimensional arrays:

```
float[,] myArr= new float[2, 3]; //Declaring a
                                //two-dimensional array
                                //of floating-point numbers
//Declaring and initializing a two-dimensional
//array of integers
int[,] myArr1 = new int[2, 3] {{ 1, 2, 3 }, { 3, 4, 6 }};
int[,] myArr2 = new int[,] {{ 1, 2, 3 }, { 3, 4, 6 }};
int[,] myArr3 = {{ 1, 2, 3 }, { 3, 4, 6 }};
```

Access to the elements of two-dimensional array is performed by specifying a line/column in the following way:

```
myArr1[1, 2] = 100; //Writing values in the third
                    //column of the second line
Console.WriteLine(myArr1[0, 0]);
                    //Output 1 (an element of the first
                    //column in the first line)
```

Examples of declaration and initialization of three-dimensional arrays are shown below:

```
int[, ,] array1 = new int[3, 4, 2]; //three-dimensional
//array of integers
//Declaring and initializing three-dimensional
//arrays of integers
int[, ,] array2 = new int[,,] {{{1, 2, 3}, {4, 5, 6}},
                               {{7, 8, 9}, {10, 11, 12}}};
int[, ,] array3 = new int[2, 2, 3] {{{1, 2, 3}, {4, 5, 6}},
                                     {{7, 8, 9}, {10, 11, 12}}};
```

Since all the arrays are inherited from the System.Array class, the methods of this class can be used when working with these arrays. Let's look at some of the System.Array class methods:

- **GetLength** returns the number of elements in the array for the specified dimension.
- **GetLowerBound** and **GetUpperBound** return the lower and upper bounds of the array for the specified dimension (for example, if there is a one-dimensional array of 5 elements, then the lower bound will be "0", and the upper will be "4").
- **CopyTo** copies all the elements of a one-dimensional array to another, starting from the specified position.
- **Clone** performs a shallow copy of the array. The copy returns as a `System.Object[]` array.
- **BinarySearch** static method performs a binary search for a value in the array (in the range of the array).
- **Clear** static method assigns the default values of element type to each element in the array.
- **IndexOf** static method returns the index of the first occurrence of the desired element in the array, returns "-1" in case of failure. The search is performed from the beginning of the array.
- **LastIndexOf** static method returns the index of the first occurrence of the desired element in the array. The search is performed from the end of the array. In case of failure it returns "-1".
- **Resize** static method changes the size of the array.
- **Reverse** static method reverses the array (array range).
- **Sort** static method performs sorting of the array (array range).

There are also extension methods:

- **Sum** performs summing of array elements.
- **Average** calculates the arithmetic mean of the array elements.
- **Contains** returns true if the specified element is present in the array.
- **Max** returns the maximum element of the array.
- **Min** returns the minimum element of the array.

And finally, there is a couple of properties:

- **Length** property returns the length of the array.
- **Rank** property returns the number of dimensions in the array.

In order to consolidate the information received, let's consider the following example:

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

    int[] myArr2 = new int[20];
    PrintArr("Array myArr2 before copying:", myArr2);
    myArr1.CopyTo(myArr2, 5);
    PrintArr("Array myArr2 after copying:", myArr2);
```

```
    Array.Clear(myArr2, 0, myArr2.GetLength(0));
    PrintArr("Array myArr2 after cleaning: ",
              myArr2);
    Array.Resize(ref myArr2, 10);
    PrintArr("Array myArr2 after resizing: ",
              myArr2);
    myArr2 = new[] { 1, 5, 3, 2, 8, 9, 6, 10, 7, 4 };
    PrintArr("Unsorted array myArr2: ", myArr2);
    Array.Sort(myArr2);
    PrintArr("Array myArr2 after sorting: ", myArr2);
    Console.WriteLine("Number 5 is located
                       in the array at" +
    Array.BinarySearch(myArr2, 5) + " the position");
    Console.WriteLine("The maximum element in the
                       array myArr2: " + myArr2.Max());
    Console.WriteLine("The minimum element in the
                       array myArr2: " + myArr2.Min());
    Console.WriteLine("The arithmetic mean of the
                       elements in myArr2: " +
                       myArr2.Average());

    int[,] myArr3 = { { 1, 2, 3 }, { 4, 5, 6 } };
    Console.WriteLine("The number of dimensions of
                       the myArr3 array: " +
                       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();
}
```

The result of the code is shown in Figure 1.4:

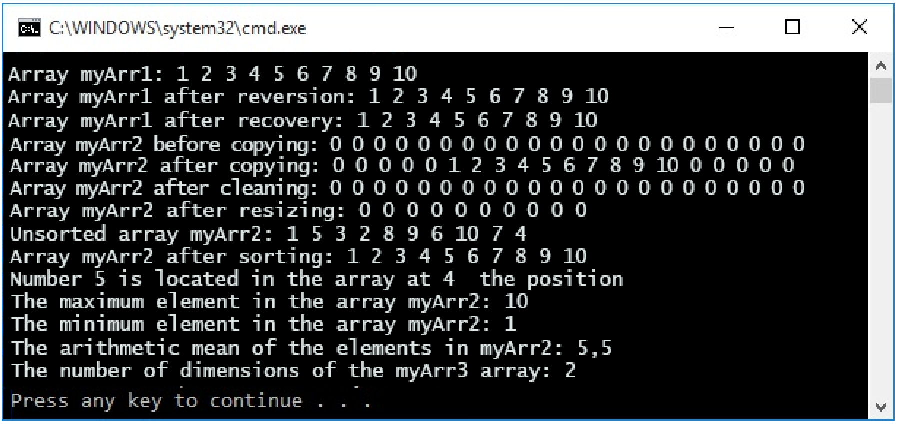


Figure 1.4. Example of working with arrays

This example illustrates the work with the array methods, as well as passing an array to a method (in our case, the `PrintArr` method).

Jagged arrays

In addition to one-dimensional and multi-dimensional arrays, C# also supports jagged arrays. The syntax for declaring such an array looks the following way:

```
Array_elements_type [][] array_name;
```

Jagged array is an array of arrays, i.e. each cell of this array contains a one-dimensional array.

```
int[][] myArr = new int[2][]; //Creating an
                              //external array
                              //of two cells
```

```
myArr[0] = new int[] {1, 2};   //Creating
                               //a one-dimensional
                               //array in
                               //the first cell
myArr[1] = new int[] {3, 4, 5, 6}; //Creating
                               //a one-dimensional
                               //array in the second
                               //cell
```

As a result, we get the following array:

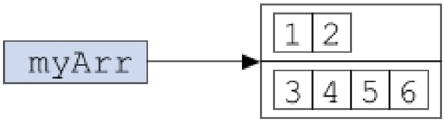


Figure 1.5. Jagged array diagram

Access to the elements of such an array is performed the following way:

```
Console.WriteLine(myArr[1][2]); //We will see 5 on
                                //the screen
```

Since working with this type of array can be difficult, let's consider an example, which demonstrates filling of an array and displaying it on the screen:

```
static void Main(string[] args)
{
    int size = 5;
    int[][] arr = new int[size][]; //Declaration of
                                   //a nested array
    for (int i = 0; i < arr.Length; i++)
    {
        arr[i] = new int[i + 1];   //creating internal
                                   //arrays
    }
```

```
    for (int i = 0; i < arr.Length; i++) //Length is
                                //the number of lines
    {
        for (int j = 0; j < arr[i].Length; j++) //Length
        //is the number of elements of the current
        //internal array
        {
           arr[i][j] = i + j + 1; //filling the internal
                                  //arrays
           Console.Write(arr[i][j] + " "); //displaying
                                  //the elements
        }
        Console.WriteLine();
     }
}
```

Program outcome (Figure 1.6):

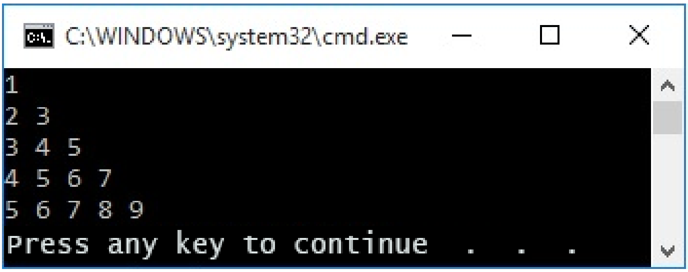


Figure 1.6. Example of working with a jagged array

Using foreach loop

In this section, we'll consider the use of «`foreach`» loop when working with arrays. As you remember, it is used for alternate iteration through a collection. This loop is convenient when working with arrays, since you would not have to enter variables to iterate through an array, take its length into account, and follow the increment, because the «foreach» loop will do this all itself (Figure 1.7).

```
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();
}
```

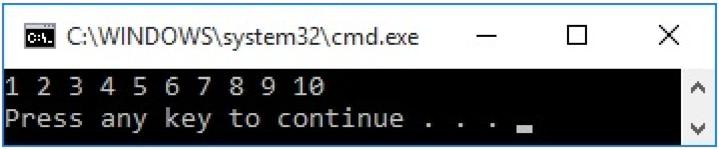


Figure 1.7. Working principle of the foreach loop

The only thing that does not allow abandoning the «`for`» and «`while`» loops in favor of «`foreach`» is that this loop works in the read-only mode, and it is impossible to change the array elements within this loop, since it will cause an error at the compile time (Figure 1.8).

```
static void Main(string[] args)
{
    foreach (int i in myArr)
    {
        i = 23; // Error
        Console.Write(i + " ");
    }
}
```

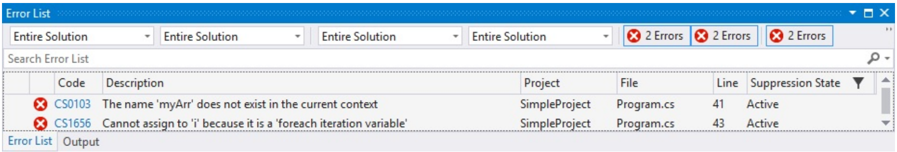


Figure 1.8. Error: changing array elements within the foreach loop

The `foreach` loop is not convenient when working with multi-dimensional arrays, because it will output all the elements of the dimension in a single line.

```
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("One-dimensional array");
    foreach (int i in myArr1)
    {
       Console.Write(i + " ");
   }
   Console.WriteLine("\nTwo-dimensional array");
   foreach (int i in myArr2)
   {
       Console.Write(i + " ");
   }
   Console.WriteLine("\nJagged array");
   for (int i = 0; i < myArr3.Length; ++i)
   {
       foreach (int j in myArr3[i])
       {
           Console.Write(j + " ");
       }
```

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

Program outcome (Figure 1.9):

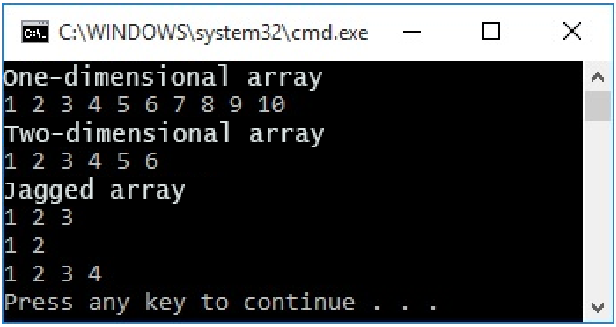


Figure 1.9. Examples of using the foreach loop with arrays

2. Strings

Creating a string

String reference data type is a sequence of zero or more characters in the Unicode and is a pseudonym for the System.String class of the .NET Framework platform.

Since strings are of the System.String type, then they are objects. This means that strings are placed in the "heap" and have a wide set of methods. Despite the fact that a string is a reference, it is convenient to create it without the new keyword.

```
string string_name = value;
```

Despite such a simple way of creating a string, the System.String class has 8 constructors, with the help of which a string is created from an array of characters. Here are examples of using such constructors.

```
string str1 = "Simple string";
char[] chrArr={'S', 'i', 'm', 'p', 'l', 'e', ' ',
               's', 't', 'r', 'i', 'n', 'g'};
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);
```

Code outcome (Figure 2.1):

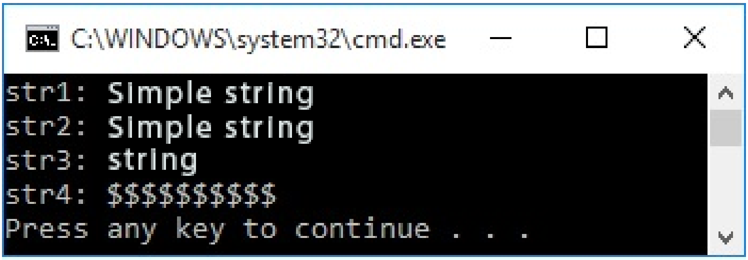


Figure 2.1. Examples of using the System.String class constructors

Just as in C++, C# has a sequence of control characters. All the characters of this sequence begin with the '\' character. Therefore, if we want to include backslashes, single or double quotes directly into the string literal, then we need to specify an additional '\' character in front of them (Figure 2.2).

```
string str = "\\"STEP\""\IT Academy\\";
Console.WriteLine(str);
```

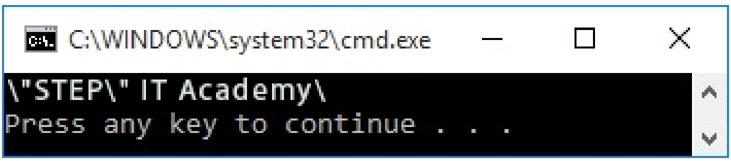


Figure 2.2. The use of control characters

Since it is often necessary to work with paths to files or folders, the concept of "verbatim" strings was introduced. The "@" character is used before these strings, and all the characters of a verbatim string are perceived literally just as they are.

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

The result is two identical outputs (Figure 2.3):

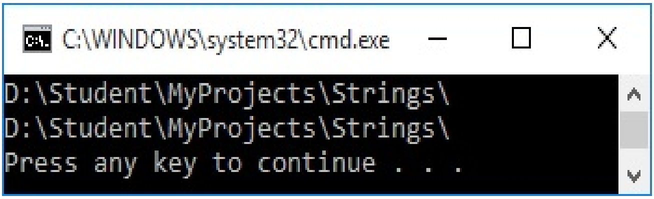


Figure 2.3. The use of "verbatim" strings

Operations with strings

The `System.String` class contains the sufficient number of methods that allow performing various operations with strings. Let's review the features of this class:

- Indexer allows the index to obtain the string character. It works in the read mode only.
- **Length** property returns the length of the string.
- **CopyTo** method copies a specified number of characters into a char type array.

```
static void Main(string[] args)
{
    string str = "Simple string";
    char[] chrArr = new char[6];
```

```
        Console.WriteLine("String reversion using
                           the indexer");
        for (int i = str.Length - 1; i >= 0; --i)
            Console.Write(str[i]);

        Console.WriteLine("\nCopying the string into
                           an array of characters");
//Copying six characters starting from the eighth
//position and putting them into an array
        str.CopyTo(8, chrArr, 0, 6);
        Console.WriteLine(chrArr);
}
```

Outcome (Figure 2.4):

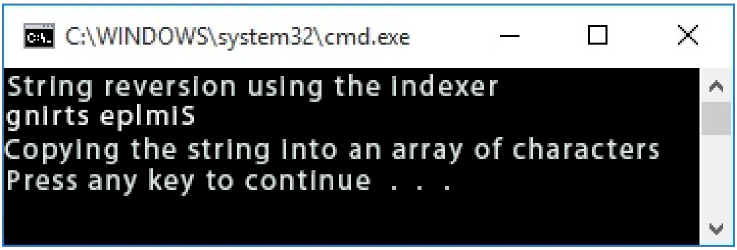


Figure 2.4. Example of using the string operations

- **Equals** method compares the values of two strings.
- **Compare** static method compares two strings passed as arguments.
- **CompareOrdinal** static method compares two strings, evaluating the numeric values of the corresponding characters in each string.
- **CompareTo** method compares the string with the string passed as a parameter. Returns an integer value: if it is less than zero, then this string precedes the parameter,

if it is zero, then the strings are equal, and if it is greater than zero, then this string follows the parameter.

- **StartsWith** method determines whether the current string begins with the specified substring.
- **EndsWith** method determines whether the current string ends with the specified substring.

```
static void Main(string[] args)
{
    string str1 = "Simple string";
    string str2 = "String";
    string str3 = "String";
    string[] strArr = {"STEP", "stepping", "running",
                       "eating", "Playing"};

    Console.WriteLine("\"" + str1 + "\" equal \"" + str2
                      + "\" : " + str1.Equals(str2));
    Console.WriteLine("\"" + str2 + "\" == \"String\"
                       : " + (str2 == "String"));
    Console.WriteLine("\"" + str2 + "\".CompareTo(\"" +
                       str3 + "\") : " + str2.
                       CompareTo(str3));
    Console.WriteLine("Insensitive comparison:");
    Console.WriteLine("\"" + str2 + "\" equal \"" +
               str3 + "\" : " + str2.Equals(str3,
               StringComparison.
               CurrentCultureIgnoreCase));
    Console.Write("Words starting with \"step\": ");
       foreach(string s in strArr)
          if(s.StartsWith("step", StringComparison.
                           CurrentCultureIgnoreCase))
                 Console.Write(s + " ");
          Console.Write("\nWords ending with
                         \"ing\": ");
```

```
            foreach(string s in strArr)
                  if(s.EndsWith("ing",
                  StringComparison.CurrentCultureIgnoreCase))
               Console.Write(s + " ");
            Console.WriteLine();
}
```

Outcome (Figure 2.5):

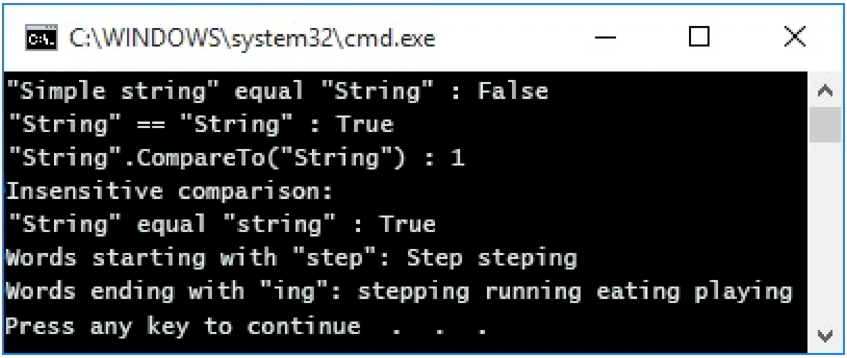


Figure 2.5. Example of the string operations

- **IndexOf** and **LastIndexOf** methods return the index of the first/last occurrence of a character/substring in the original string.
- **IndexOfAny** and **LastIndexOfAny** methods return the index of the first/last occurrence of any listed character in the original string.
- **Substring** method returns a substring from the current string.

All the search methods include overload versions for searching in the specified range with the specified comparison method.

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

     Console.WriteLine("The first occurrence of the
                       character \'I\': " +
                       str1.IndexOf('I'));
     Console.WriteLine("The first occurrence of the
                       substring \"Inheritance\" : " +
                       str1.IndexOf('I'));
     Console.WriteLine("The last occurrence of the
                       character \'E\': " +
                       str1.LastIndexOf('E'));
     Console.WriteLine("The last occurrence of any
                       character in the string" +
                       "\"ABCDEFGHIJKLMN\" : " +
                       str1.LastIndexOfAny(str2.
                       ToCharArray()));
     Console.WriteLine("Substring starting from the
                       11th character up to 23rd : " +
                       str1.Substring(11, 12));
}
```

Outcome (Figure 2.6):

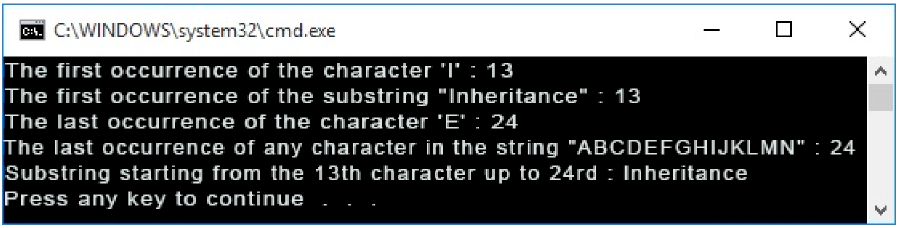


Figure 2.6. Example of using the string operations

- **Concat** static method performs concatenation. A convenient alternative to this method is "+" and "+ =" operations.

- **ToLower** and **ToUpper** methods return a string in the lower and upper case, respectively.
- **Replace** method replaces all the occurrences of a character/substring to the specified character/substring.
- **Contains** method checks whether the specified character/substring is included in the original string.
- **Insert** method inserts a substring at the given position of the original string.
- **Remove** method removes all the occurrences of the specified substring from the current string.
- **PadLeft** and **PadRight** methods add the specified characters to the left/right of the original string. If the character is not specified, the space character will be added. The first parameter indicates the number of characters in the string, i.e. the total number of characters after padding.
- **Split** method splits a string into substrings by the specified separator characters. Returns an array of resulting strings. In order to exclude empty strings from the array, it is necessary to use this method with the `StringSplitOptions.RemoveEmptyEntries` parameter.
- **Join** static method combines strings of the specified array into a single string, and alternates them with the specified separator character.
- **TrimLeft** and **TrimRight** methods clean spaces (by default) or certain characters from the beginning or the end of the string, respectively. **Trim** method does the same on both sides of the string.

And now let's consider an example of the methods listed above:

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

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

    str4 = str4.Replace("learn", "study");
    Console.WriteLine(str4);

    str4 = str4.Insert(4, "hard") .ToUpper();
    Console.WriteLine(str4);

    if (str4.Contains("hard"))
        Console.WriteLine("I learn really hard :)");
    else
        Console.WriteLine("I learn the best I can");

    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 += "learn";
    Console.WriteLine(str4);
}
```

Outcome (Figure 2.7):

```
C:\WINDOWS\system32\cmd.exe
I + learn + C# = I learn C#
I study C#
I study C# hard
I learn the best I can
I study C# hard
I study C# hard
I
study
C#
hard
I learn hard
Press any key to continue....
```

Figure 2.7. Example of using the string operations

- **Format** static method allows convenient formatting of the string. The first parameter is a format string, which contains a text displayed on the screen. If it is necessary to insert values of variables to this string, the insertion point is marked by the index in the braces. If necessary, here can also be specified the number of symbols occupied by the inserted element and its format specifier. The inserted data itself is indicated with the following method parameters. Thus, the syntax of using the Format method is the following:

```
String.Format("Printed Text {index,
              size:specifier}", data);
```

The format specifiers:

1. "**C**" or "**c**" — for numeric data. Displays a local currency symbol.
2. "**D**" or "**d**" — for integer data. Displays a normal integer.

3. "**E**" or "**e**" — for numeric data. Displays a number in an exponential form.
4. "**F**" or "**f**" — for numeric data. Displays a number with a fixed decimal point.
5. "**G**" or "**g**" — for numeric data. Displays a usual number.
6. "**N**" or "**n**" — for numeric data. Displays a number in a local settings format.
7. "**P**" or "**p**" — for numeric data. Displays a number with the percent symbol '%'.
8. "**X**" or "**x**" — for integer data. Displays a number in hexadecimal format.

```
double test1=99989.987;
int test2 = 99999;

Console.WriteLine(String.Format("c format: {0,15:C}",
                                 test1));
Console.WriteLine(String.Format("D format: {0:D9}",
                                 test2));
Console.WriteLine(String.Format("E format: {0:E}",
                                 test1));
Console.WriteLine(String.Format("f format: {0:F2}",
                                 test1));
Console.WriteLine(String.Format("G format: {0:G}",
                                 test1));
Console.WriteLine(String.Format("N format: {0,15:N}",
                                 test2));
Console.WriteLine(String.Format("P format: {0:P}",
                                 test1));
Console.WriteLine(String.Format("X format: {0:X}",
                                 test2));
Console.WriteLine(String.Format("x format: {0:x}",
                                 test2));
```

Outcome (Figure 2.8):

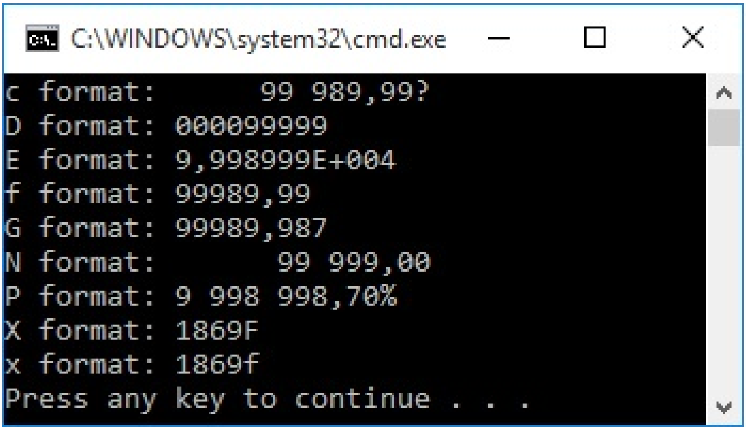


Figure 2.8. Example of using the Format method

In the C# version 6.0, a new concept of **interpolated strings** was introduced. In fact, it makes possible formatting strings without using the **Format** method, but with the same outcome. To do this, the '$' symbol should precede the string, and a variable or a value to be inserted into the string should be specified in the braces in the string itself. It also became possible to use a ternary operator in the string. Examples of program code and outcome (Figure 2.9) are demonstrated below.

```
int number1 = 56, number2=45;
Console.WriteLine(String.Format("Number 1 is equal
                  to {0}. Number 2 is equal to {1}.",
                  number1,number2));
Console.WriteLine($"number 1 is equal to {number1}.
                  Number 2 is equal to {number2}.");
   //using the ternary operator
Console.WriteLine($"Number 1 {(number1 > number2 ?
                  "is more than":" is less than");
```

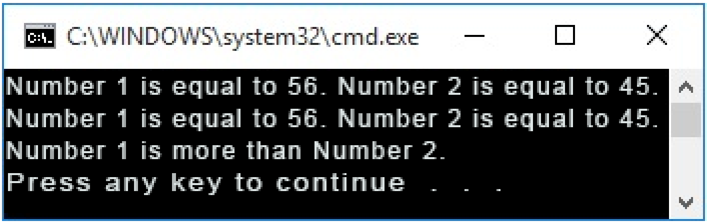


Figure 2.9. Examples of using the interpolated lines

Features of using the strings

When working with the strings, it is necessary to consider the fact that the strings are immutable in C#, i.e. it is impossible to make any changes in a string without recreating it. But don't worry about it — a string is created and destroyed automatically, you only need to accept a reference to it and continue working. It should be understood that the reference variables of the `string` type can change the objects, which they refer to. The content of the created string object is no longer possible to change. Example and outcome (Figure 2.10) are shown below.

```
static void Main(string[] args)
{
    string str1 = "Original string";
    Console.WriteLine(str1);
    str1 += " (but in another memory location)";
    Console.WriteLine(str1);
}
```

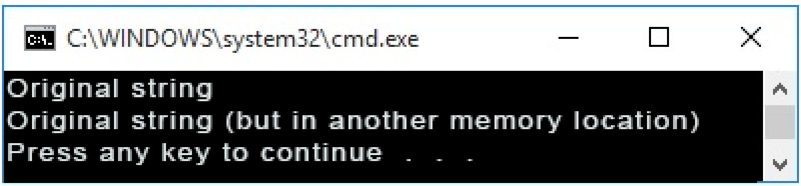


Figure 2.10. Features of the string type

In the above example, we first create a string of 15 characters. But when trying to add another string to it, a new memory is allocated for a string of 46 characters, to which the content of both originals is copied. The old string undergoes "garbage collection" (which will be considered in the corresponding lesson).

This feature of the strings leads to the fact that frequent modification of a string requires a lot of resources (both memory and "garbage collector", which may cause lower performance).

StringBuilder class was created in order to avoid performance losses. This class has a less extensive set of methods compared to the String class, but allows working with an object located in the same place in memory. The memory is redistributed only when the object of **StringBuilder** type lacks space for the changes made. The maximum number of characters that can be allocated in the memory is doubled.

Let us briefly consider the methods of the **StringBuilder** class.

- **Append** adds data of any of the standard types to the original string.
- **AppendFormat** adds a string formed in accordance with the format specifiers to the original string.
- **Insert** inserts data of any of the standard types in the original string.
- **Remove** removes a range of characters from the original string.
- **Replace** replaces a character/substring in the original string with the specified character/string.

- **CopyTo** copies the characters of the original string into a char array.
- **ToString** converts a StringBuilder object into a String object.

Also, the following properties exist in the `StringBuilder` class:

- **Length** returns the number of characters that are present in the string at the moment.
- **Capacity** returns or sets the number of characters that can be placed in a string without allocating additional memory.
- **MaxCapacity** returns the maximum string capacity.

The `StringBuilder` class has a great advantage over the `String` class when intensively working with strings, so it is recommended to be used in the situations like this. In other cases, the `String` class is more convenient.

Example and outcome (Figure 2.11) of working with the `StringBuilder` class are shown below.

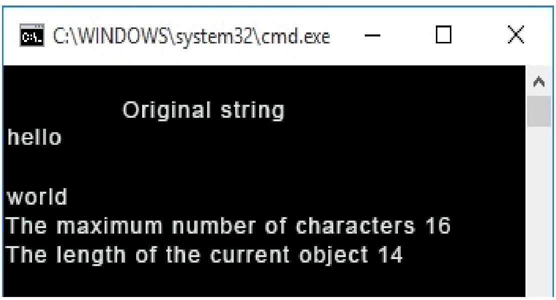
```
StringBuilder sb = new StringBuilder();
//sb = "hello"; Error
sb.Append("hello"); //adding a string to the existing one
sb.AppendLine();      //adding an empty string to
                      //the existing one
sb.AppendLine();
sb.Append("world");

Console.WriteLine("\n\tOriginal string");
Console.WriteLine(sb);
Console.WriteLine("The maximum number of characters " +
                  sb.Capacity);
Console.WriteLine("The length of the current object " +
                  sb.Length);
```

```
Console.WriteLine("\n\tInserting a string");
sb.Insert(7, "abracadabra"); //inserting a string at
                             //the given position
Console.WriteLine(sb);
Console.WriteLine("The maximum number of characters " +
                  sb.Capacity);
Console.WriteLine("The length of the current object " +
                  sb.Length);

Console.WriteLine("\n\tReplacing 'a' characters
                  with 'z'");
sb.Replace('a', 'z'); //replacing the characters
                      //of a string
Console.WriteLine(sb);

Console.WriteLine("\n\tRemoving 10 characters
                   starting from 3");
sb.Remove(3, 10); //Removing characters from a string
Console.WriteLine(sb);
Console.WriteLine("The maximum number of characters " +
                   sb.Capacity);
Console.WriteLine("The length of the current object " +
                   sb.Length);
```



```
          Inserting a string
hello
abracadabra
world
The maximum number of characters 27
The length of the current object 25

        Replacing 'a' characters with 'z'
hello
zbrzczdzbrz
world

      Removing 10 characters starting from 3
heldzbrz
world
The maximum number of characters 19
The length of the current object 15
Press any key to continue . . .
```

Figure 2.11. The results of working with the StringBuilder class

It should be noted that initialization of the StringBuilder class with a string will cause an error at the compile time (Figure 2.12).

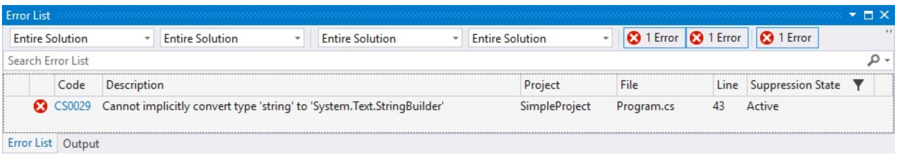


Figure 2.12. StringBuilder class initialization error

3. Using the command line arguments

Just as in other programming languages, the command line parameters can be transmitted to the C# program. There is only one parameter for working with the command line parameters in the `Main` method — `args` of the `string[]` type. This parameter contains the command line arguments transmitted to the program.

In C++, a full name of the executable (i.e. file name) was passed by a zero command line parameter. In C #, the zero parameter passes the first command line parameter. In other words, the `args` array contains command line parameters only.

To make the work with the command line easier while debugging, the necessary parameters can be entered during the design process (to run the program via the IDE). To do this, go to the project properties and select the «Debug» tab, then enter the required parameters in the «Command line arguments» window, as illustrated in the Figure 3.1.

The `System.Environment.CommandLine` object is used for obtaining a full path of the console program executable.

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

Let's consider outcome (Figure 3.2) of command line parameters output:

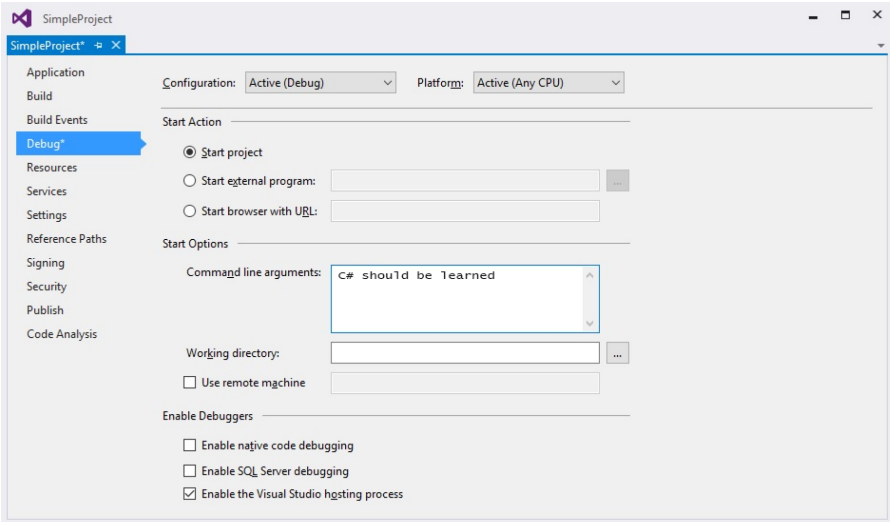


Figure 3.1. Entering command line parameters via the IDE

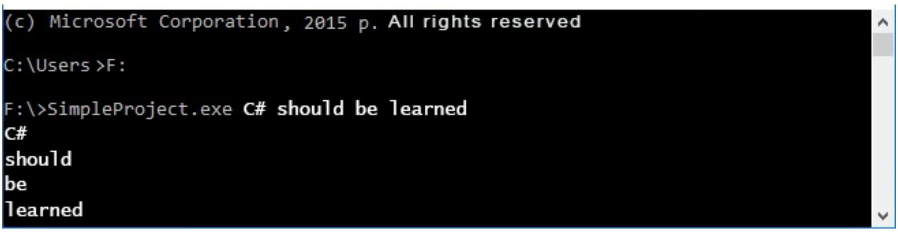


Figure 3.2. The output of command line parameters

4. Enumerations (enum)

The concept of enumeration

Enumeration is a non-empty list of named constants. It sets all the values that a variable of the type can take. Enumerations are a class and are inherited from the `System.Enum base class`.

The syntax of enumeration declaration

The `enum` keyword is used to declare enumeration, a name of enumeration is specified, and names of constants are listed in the braces:

```
enum EnumName {elem1, elem2, elem3, elem4}
```

A good example of enumeration is days of the week:

```
enum DayOfWeek
{
    Monday, Tuesday, Wednesday, Thursday, Friday,
    Saturday, Sunday
}
```

The following example of enumeration describes the possible types of trucks:

```
enum TransportType
{
    Semitrailer, Coupling, Refrigerator,
    OpenSideTruck, Tank
}
```

The enumerated elements can be addressed by specifying a name of the enumeration class and via a name point of the particular element of the enumeration.

```
DayOfWeek day = DayOfWeek.Monday;
```

Enumeration constants are of integer type int. By default, the first constant has the value of 0, and the value of every following constant is incremented by 1. In this example of the `DayOfWeek` enumeration, values of the constants will be as follows: Monday = 0, Tuesday = 1, Wednesday = 2, Thursday = 3, Friday = 4, Saturday = 5, Sunday = 6. Arithmetic operations can be performed with the variables of enum types. Here is an example of the method that returns the next day of the week:

```
public DayOfWeek NextDay(DayOfWeek day)
{
   return (day < DayOfWeek.Sunday) ? ++day : DayOfWeek.Monday;
}
```

You can also assign a value to a constant explicitly. For example, the following enumeration describes amount of a discount for different types of customers:

```
enum Discount
{
   Default, Incentive = 2, Patron = 5, VIP = 15
}
```

The need and features of applying enumeration

Enumerations allow making programming easier and faster, helping to get rid of the confusion when assigning values to variables. Here are useful features of enumerations:

- enumerations ensure that only permitted, expected values will be assigned to the variables. If you try to assign the value that is not included in the list of acceptable values to an instance of enumeration, the compiler will generate an error;
- enumerations make a code easier to understand, since we are referring not just to numbers, but to some meaningful names;
- enumerations allow saving programmer's time. When you want to assign a certain value to an instance of enum type, then the IntelliSense environment integrated into the Visual Studio will display a list of all possible values;
- as already mentioned, enumerations are inherited from the `System.Enum` base class, that allows calling a range of useful methods for them (it will be discussed in more detail below).

In general, enumerations are used when it is necessary to perform certain actions, based on matching a value of enumeration with possible values using the `switch` statement.

Consider an example wherein the user selects a commodity name from a list, and a program determines the best vehicle for its transportation. The program code is shown below:

```
enum CommodityType //commodity type
{
    FrozenFood, Food, DomesticChemistry,
    BuildingMaterials, Petrol
}

enum TransportType //vehicle type
{
```

```
    Semitrailer, Coupling, Refrigerator,
    OpenSideTruck, FuelTruck
}

static void Main(string[] args)
{
    Console.WriteLine("Enter a number from 1 to 5");

    int number = Int32.Parse(Console.ReadLine());

    if (number > 0 && number < 6)
    {
        CommodityType commodity = (CommodityType)Enum.
        GetValues(typeof(CommodityType)).
        GetValue(number - 1);

        TransportType transport = TransportType.
                                  Semitrailer;
        switch (commodity)
        {
            case CommodityType.FrozenFood:
                transport = TransportType.
                            Refrigerator;
                break;
            case CommodityType.Food:
                transport = TransportType.Semitrailer;
                break;
            case CommodityType.DomesticChemistry:
                transport = TransportType.Coupling;
                break;
            case CommodityType.BuildingMaterials:
                transport = TransportType.OpenSideTruck;
                break;
            case CommodityType.Petrol:
                transport = TransportType.FuelTruck;
                break;
        }
```

```
            Console.WriteLine("For the commodity - {0}
                              the best vehicle is - {1}.",
                              commodity, transport);
        }
        else
        {
            Console.WriteLine("Input error");
        }
}
```

CommodityType and TransportType enumerations describe types of commodities and types of trucks, respectively. Based on the information entered by the user, and after its validation, we get a value from the CommodityType enumeration. To do this, we use the GetValues method of the System.Enum base class, which returns the System.Array class, the GetValue method of which returns the value of the object by the index. Then we compare the types of commodities and assign the appropriate type of transport. One of the possible options of the program is shown in the Figure 4.1:

Figure 4.1. Possible option of the program with the use of enumerations

Installing the base enumeration type

The base type refers to the type of enumeration constants. As already mentioned, enumerations are based on the int type by default. But it is possible to create an enumeration based on any of the integer types: `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`. To do this, when declaring an enumeration, the desired type is specified following its name and colon:

```
enum NameEnum [:baseType] {EnumList}
```

For example, let's consider an enumeration that describes an average distance between the planets of the solar system and the sun in kilometers:

```
enum DistanceSun : ulong
{
    Sun = 0,
    Mercury = 57900000,
    Venus = 108200000,
    Earth = 149600000,
    Mars = 227900000,
    Jupiter = 7783000000,
    Saturn = 1427000000,
    Uranus = 2870000000,
    Neptune = 4496000000,
    Pluto = 5946000000
}
```

Using the methods for enumerations

Enumerations are a class inherited from the `System.Enum` base class. This means that the methods of this class can be used for them, i.e. the methods of comparing the values of enumeration, conversion of the values of enumeration into the string representation, methods of converting the string representation of the value into an enumeration, etc. Here are some of them:

- **CompareTo** compares the current instance with the specified object and returns the value less than 0 if the current instance is less than specified, 0 if the values are equal, the value greater than 0 if the current instance is greater than the specified value.

- **GetName** displays a name of the constant in the specified enumeration, which has a predetermined value.
- **GetNames** static method displays names of the array constants in the specified enumeration.
- Static method **GetValues** displays the array of constant values in the specified enumeration.
- **IsDefined** static method returns the presence bit of the constant with the specified value in the specified enumeration. Returns `true`, if the constant is present, otherwise returns `false`.
- **Parse** static method converts the string representation of the name or numeric value of one or more enumerated constants into an equivalent enumerated object.
- **ToString** converts the value of this instance into its equivalent string representation.

Example of the `System.Enum` class methods when working with the enumerations is demonstrated below:

```
enum DistanceSun : ulong
{
    Sun = 0, Mercury = 57900000, Venus = 108200000,
    Earth = 149600000,
    Mars = 227900000, Jupiter = 7783000000,
    Saturn = 1427000000,
    Uranus = 2870000000, Neptune = 4496000000,
    Pluto = 5946000000
}
```

```
static void Main(string[] args)
{
    string moon = "Moon";
    //verification of the presence of the constants
    //in the specified enumeration
    if (!Enum.IsDefined(typeof(DistanceSun), moon))
    {
        Console.WriteLine("\tValue " + moon +
                          "unavailable in
                          the enumeration.");
    }
    Console.WriteLine("\n\tFormatted output of all
                      the values of the constants of
                      this enumeration.");
    foreach (DistanceSun item in Enum.
             GetValues(typeof(DistanceSun)))
    {
        Console.WriteLine("{0,-10} {1,-10} {2,20}",
        Enum.Format(typeof(DistanceSun), item, "G"),
       //output as a string with the name of the constant
        Enum.Format(typeof(DistanceSun), item, "D"),
        //output in the form of a decimal value
        Enum.Format(typeof(DistanceSun), item, "X"));
        //output in the form of a hexadecimal value
    }
    Console.WriteLine("\n\tAll the values of the
                      constants of the specified
                      enumeration.");

    foreach (string str in Enum.
             GetNames(typeof(DistanceSun)))
    {
        Console.WriteLine(str);
    }
   ulong number = 227900000;
```

```
        Console.WriteLine("\n\tConstant name with
                          the value of {0} from the
                          specified enumeration.\n",
                          number);
        Console.WriteLine(Enum.
                          GetName(typeof(DistanceSun),
                          number));
      }
    }
}
```

Code uotcome (Figure 4.2):

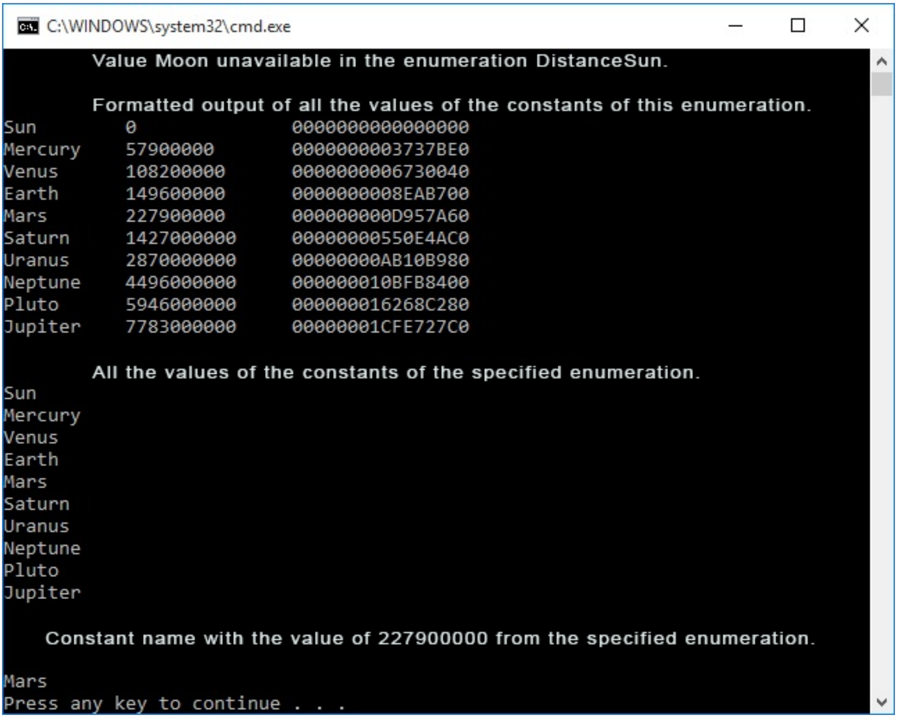


Figure 4.2. The work of the program with the use of enumerations

Home task

1. Compress an array, deleting all zeros from it, and then fill the elements available from the right with the values of -1.
2. Convert the array so that the negative elements are placed first following by positive values (consider 0 as positive).
3. Write a program that offers the user to enter a number, and then counts how many times this number occurs in the array.
4. Swap the columns of the two-dimensional array of M by N.

Lesson 2

Arrays and strings. Enumerations

© Yuriy Zaderey
© STEP IT Academy.
www.itstep.org

All rights to protected pictures, audio, and video belong to their authors or legal owners.
Fragments of works are used exclusively in illustration purposes to the extent justified by the purpose as part of an educational process and for educational purposes in accordance with Article 1273 Sec. 4 of the Civil Code of the Russian Federation and Articles 21 and 23 of the Law of Ukraine "On Copyright and Related Rights". The extent and method of cited works are in conformity with the standards, do not conflict with a normal exploitation of the work, and do not prejudice the legitimate interests of the authors and rightholders. Cited fragments of works can be replaced with alternative, non-protected analogs, and as such correspond the criteria of fair use.

All rights reserved. Any reproduction, in whole or in part, is prohibited. Agreement of the use of works and their fragments is carried out with the authors and other right owners. Materials from this document can be used only with resource link.

Liability for unauthorized copying and commercial use of materials is defined according to the current legislation of Ukraine.