Chapter 11. Creating and Using Objects

In This Chapter

In this chapter we are going to get familiar with the basic concepts of object-oriented programming – **classes and objects** – and we are going to explain how to use classes from the standard libraries of .NET Framework. We are going to mention some commonly used system classes and see how to **create and use** their instances (objects). We are going to discuss how we **can access fields** of an object, how to **call constructors** and how to work with static fields in classes. Finally, we are going to get familiar with the term "**namespaces**" – how they help us, how to include them and use them.

Classes and Objects

Over the last few decades programming and informatics have experienced incredible growth and concepts, which have changed the way programs, are built. **Object-oriented programming (OOP)** introduces such radical idea. We are going to make a short introduction to the principles of OOP and the concepts used in it. Firstly, we are going to explain what classes and objects are. These two terms are basic for OOP and inseparable part from the life of any modern programmer.

What Is Object-Oriented Programming?

Object-oriented programming (OOP) is a programming paradigm, which uses **objects** and their interactions for building computer programs. Thus an easy to understand, simple model of the subject area is achieved, which gives an opportunity to the programmer to solve intuitively (by simple logic) many of the problems, which occur in the real world.

For now we are not going to get into details what the goals and the advantages of OOP are, as well as explaining in details the principles for building hierarchies of classes and objects. We are going to mention only that programming techniques of OOP often include **encapsulation**, **abstraction**, **polymorphism** and **inheritance**. These techniques are out of the goals of the current chapter and we are going to consider them later in the chapter "Principles of Object-Oriented Programming". Now we will focus on **objects** as a basic concept in OOP.

What Is an Object?

We are going to introduce the concept **object** in the context of OOP. Software objects model real world objects or abstract concepts (which are also regarded as objects).

Examples of **real-world objects** are people, cars, goods, purchases, etc. abstract objects are concepts in an object area, which we have to model and use in a computer program. Examples of abstract objects are the data structures stack, queue, list and tree. They are not going to be a subject in this chapter, but we are going to see them in details in the next chapters.

In objects from the real world (as well as in the abstract objects) we can distinguish the following two groups of their characteristics:

- **States** – these are the characteristics of the object which define it in a way and describe it in general or in a specific moment
- **Behavior** – these are the specific distinctive actions, which can be done by the object.

Let's take for example an object from the real world – "dog". The states of the dog can be "name", "fur color" and "breed", and its behavior – "barking", "sitting" and "walking".

Objects in OOP combine data and the means for their processing in one. They correspond to objects in real world and contain data and actions:

- **Data members** – embedded in objects variables, which describe their states.
- **Methods** – we have already considered them in details. They are a tool for building the objects.

What Is a Class?

The **class** defines abstract characteristics of objects. It provides a structure for objects or a pattern which we use to describe the nature of something (some object). **Classes are building blocks of OOP** and are inseparably related to the **objects**. Furthermore, each object is an **instance** of exactly one specific class.

We are going to give as an **example a class and an object**, which is its instance. We have a **class `Dog`** and an **object `Lassie`**, which is an instance of the class **`Dog`** (we say it is an object of type **`Dog`**). The class **`Dog`** describes the characteristics of all dogs whereas **`Lassie`** is a certain dog.

Classes provide **modularity** in object-oriented programs. Their characteristics have to be meaningful in a common context so that they could be understood by people who are familiar with the problem area and are not programmers. For instance, the class **`Dog`** cannot have (or at least should not) a characteristic "RAM" because in the context of this class such characteristic has no meaning.

Classes, Attributes and Behavior

The class defines the **characteristics of an object** (which we are going to call **attributes**) and its **behavior** (actions that can be performed by the object). The attributes of the class are defined as its own variables in its body (called **member variables**). The behavior of objects is modeled by the definition of **methods** in classes.

We are going to illustrate the foregoing explanations through an **example of a real-world definition of a class**. Let's return to the example with the dog. We would like to define a class **Dog** that models the real object "dog". The class is going to include characteristics which are common for all dogs (such as breed and fur color), as well as typical for the dog behavior (such are barking, sitting, walking). In this case we are going to have attributes **breed** and **furColor**, and the behavior is going to be implemented by the methods **Bark()**, **Sit()** and **Walk()**.

Objects – Instances of Classes

From what has been said till now we know that each object is an instance of just one class and is created according to a pattern of this class. Creating the object of a defined class is called **instantiation** (creation). The **instance** is the object itself, which is created runtime.

Each object is in **instance** of a specific class. This instance is characterized by **state** – set of values, associated with class attributes.

In the context of such behavior the object consists of two things: current **state** and **behavior** defined in the class of the object. The state is specific for the instance (the object), but the behavior is common for all objects which are instances of this class.

Classes in C#

So far we have considered several common characteristics of OOP. A great part of the **modern programming languages are object-oriented**. Each of them has particular features for working with classes and objects. In this book we are going to focus only one of these languages – C#. It is good to know that the knowledge of OOP in C# would be useful to the reader no matter which object-oriented language he uses in practice. That is because **OOP is a fundamental concept in programming**, used by virtually all modern programming languages.

What Are Classes in C#?

A **class** in C# is defined by the keyword **class**, followed by an identifier (name) of the class and a set of data members and methods in a separate code block.

Classes in C# can contain the following elements:

- **Fields** – member-variables from a certain type;
- **Properties** – these are a special type of elements, which extend the functionality of the fields by giving the ability of extra data management when extracting and recording it in the class fields. We are going to focus on them in the chapter "Defining Classes";
- **Methods** – they implement the manipulation of the data.

An Example Class

We are going to give an example of a class in C#, which contains the listed elements. The class **Cat** models the real-world object "cat" and has the properties **name** and **color**. The given class defines several fields, properties and methods, which we are going to use later. You can now see the definition of the class (we are not going to consider in details the definition of the classes – we are going to focus on that in the chapter "Defining Classes"):

```
public class Cat
{
   // Field name
   private string name;
   // Field color
   private string color;

   public string Name
   {
      // Getter of the property "Name"
      get
      {
         return this.name;
      }
      // Setter of the property "Name"
      set
      {
         this.name = value;
      }
   }

   public string Color
   {
      // Getter of the property "Color"
      get
      {
         return this.color;
      }
      // Setter of the property "Color"
      set
      {
         this.color = value;
      }
   }

   // Default constructor
   public Cat()
   {
      this.name = "Unnamed";
      this.color = "gray";
   }

   // Constructor with parameters
   public Cat(string name, string color)
   {
      this.name = name;
      this.color = color;
   }

   // Method SayMiau
   public void SayMiau()
   {
      Console.WriteLine("Cat {0} said: Miauuuuuu!", name);
   }
}
```

The example class **Cat** defines the **properties Name** and **Color**, which keep their values in the hidden (private) **fields name** and **color**. Furthermore, two **constructors** are defined for creating instances of the class **Cat**, respectively with and without parameters, and a **method** of the class **SayMiau()**.

After the example class is defined we can now use it in the following way:

```
static void Main()
{
   Cat firstCat = new Cat();
   firstCat.Name = "Tony";
   firstCat.SayMiau();

   Cat secondCat = new Cat("Pepy", "red");
   secondCat.SayMiau();
   Console.WriteLine("Cat {0} is {1}.",
      secondCat.Name, secondCat.Color);
}
```

If we execute the example, we are going to get the following output:

```
Cat Tony said: Miauuuuuu!
Cat Pepy said: Miauuuuuu!
Cat Pepy is Red.
```

We saw a simple example for defining and using classes, and in the section "Creating and Using Objects" we are going to explain in details how to create objects, how to access their properties and how to call their methods and this is going to allow us to understand how this example works.

System Classes

Calling the method **Console.WriteLine(…)** of the class **System.Console** is an example of usage of a **system class** in C#. We call system classes the classes defined in **standard libraries** for building applications with C# (or another programming language). They can be used in all our .NET applications (in particular those written in C#). Such are for example the classes **String**, **Environment** and **Math**, which we are going to consider later.

As we already know from chapter "Introduction to Programming" the **.NET Framework SDK** comes with a set of programming languages (like C# and VB.NET), compilers and **standard class library** which provides thousands of system classes for accomplishing the most common tasks in programming like console-based input / output, text processing, collection classes, parallel execution, networking, database access, data processing, as well as creating Web-based, GUI and mobile applications.

It is important to know that the implementation of the logic in classes is **encapsulated** (hidden) inside them. For the programmer it is important what they do, not how they do it and for this reason a great part of the classes is not publicly available (**public**). With system classes the implementation is often not available at all to the programmer. Thus, new **layers of abstraction** are created which is one of the basic principles in OOP.

We are going to pay special attention to system classes later. Now it is time to get familiar with creating and using objects in programs.

Creating and Using Objects

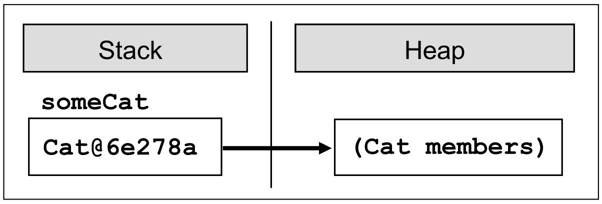
For now we are going to focus on **creating and using objects** in our programs. We are going to work with already defined classes and mostly with system classes from .NET Framework. The specificities of defining our own classes we are going to consider later in the chapter "Defining Classes".

Creating and Releasing Objects

The creation of objects from preliminarily defined classes during program execution is performed by the **operator new**. The newly created object is usually assigned to the variable from type coinciding with the class of the object (this, however, is not mandatory – read chapter "Principles of Object-Oriented Programming"). We are going to note that in this assignment the object is not copied, and only a **reference** to the newly created object is recorded in the variable (its address in the memory). Here is a simple example of how it works:

```
Cat someCat = new Cat();
```

The variable **someCat** of type **Cat** we assign the newly created **instance** of the class **Cat**. The variable **someCat** remains in the **stack**, and its value (the instance of the class **Cat)** remains in the **managed heap**:



Creating Objects with Set Parameters

Now we are going to consider a slightly different variant of the example above in which we set parameters when creating the object:

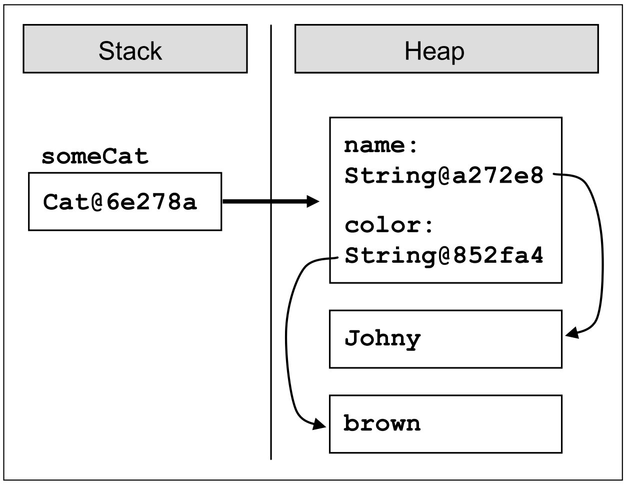
```
Cat someCat = new Cat("Johnny", "brown");
```

In this case we would like the objects **someCat** to represent a cat whose name is "**Johnny**" and is brown. We indicate this by using the words "**Johnny**" and "**brown**", written in the brackets after the name of the class.

When creating an object with the operator **new**, two things happen: memory is set aside for this object and its data members are initialized. The **initialization** is performed by a special method called **constructor**. In the example above the initializing parameters are actually parameters of the constructor of the class.

We are going to discuss constructors after a while. As the member variables **name** and **color** of the class **Cat** are of reference type (of the class **String**), they are also recorded in the **dynamic memory (heap)** and in the object itself are kept their references (addresses / pointers).

The following figure illustrates how the **Cat** object is represented in the computer memory (arrows illustrated the **references** from one object to another):

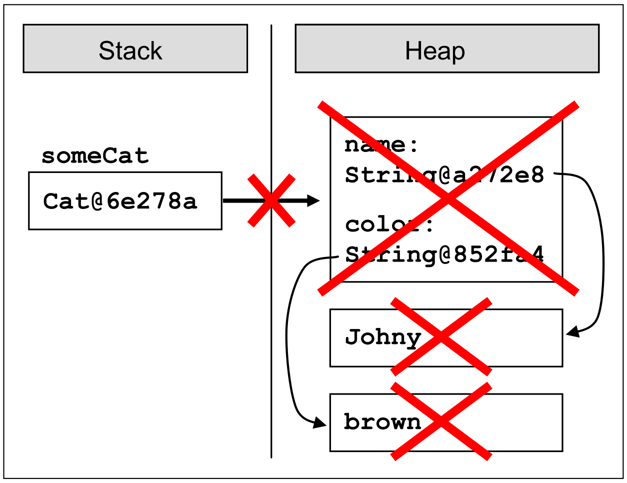


Releasing the Objects

An important feature of working with objects in C# is that usually there is no need to manually destroy them and release the memory taken up by them. This is possible because of the embedded in .NET CLR system for cleaning the memory (**garbage collector**) which takes care of releasing unused objects instead of us. Objects to which there is no reference in the program at certain moment are **automatically released** and the memory they take up is released. This way many potential bugs and problems are prevented. If we would like to manually release a certain object, we have to destroy the reference to it, for example this way:

```
someCat = null;
```

This does not destroy the object immediately, but puts it in a state in which it is inaccessible to the program and the next time the garbage collector cleans the memory it is going to be released:



Access to Fields of an Object

The **access to the fields** and properties of a given object is done by the **operator .** (dot) placed between the names of the object and the name of the field (or the property). The operator **.** is not necessary in case we access field or property of given class in the body of a method of the same class.

We can **access** the **fields** and the **properties** either to extract data from them, or to assign new data. In the case of a property the access is implemented in exactly the same way as in the case of a field – C# give us this ability. This is achieved by the keywords **get** and **set** in the definition of the property, which perform respectively extraction of the value of the property and assignment of a new value. In the definition of the class **Cat** (given above) the properties are **Name** and **Color**.

Access to the Memory and Properties of an Object – Example

We are going to give an example of using a property of an object, as well as using the already defined above class **Cat**. We create an instance **myCat** of the class **Cat** and assign **"Alfred"** to the property **Name**. After that we print on the standard output a formatted string with the name of our cat. You can see an implementation of the example:

```
class CatManipulating
{
  static void Main()
  {
    Cat myCat = new Cat();
    myCat.Name = "Alfred";

    Console.WriteLine("The name of my cat is {0}.",
      myCat.Name);
  }
}
```

Calling Methods of Objects

Calling the methods of a given object is done through the **invocation operator ()** and with the help of the **operator .** (dot). The operator **dot** is not obligatory only in case the method is called in the body of another method of the same class. Calling a method is performed by its name followed by **()** or **(<parameters>)** for the case when we pass it some arguments. We already know how to invoke methods from the chapter "Methods".

Now is the moment to mention the fact that methods of classes have **access modifiers public**, **private** or **protected** with which the ability to call them could be restricted. We are going to consider these modifiers in the chapter "Defining Classes". For now it enough to know that the access modifier

public does not introduce any restrictions for calling the method, i.e. makes it publicly available.

Calling Methods of Objects – Example

We are going to complement the example we already gave as we call the method **SayMiau** of the class **Cat**. Here is the result:

```
class CatManipulating
{
  static void Main()
  {
    Cat myCat = new Cat();
    myCat.Name = "Alfred";

    Console.WriteLine("The name of my cat is {0}.",myCat.Name);
    myCat.SayMiau();
  }
}
```

After executing the program above the following text is going to be printed on the standard output:

```
The name of my cat is Alfred.
Cat Alfred said: Miauuuuuu!
```

Constructors

The **constructor** is a special method of the class, which is called automatically when **creating an object** of this class, and performs initialization of its data (this is its purpose). The constructor has no type of returned value and its name is not random, and mandatorily coincides with the class name. The constructor can be **with or without parameters**. A constructor without parameters is also called **parameterless constructor**.

Constructor with Parameters

The constructor can **take parameters** as well as any other method. Each class can have different count of constructors with one only restriction – the count and type of their parameters have to be different (different signature). When creating an object of this class, one of the constructors is called.

In the presence of several constructors in a class naturally occurs the question which of them is called when the object is created. This problem is solved in a very intuitive way as with methods. The appropriate constructor is chosen automatically by the compiler according to the given set of parameters when creating the object. We use the principle of the **best match**.

Calling Constructors – Example

Lets' take a look again at the definition of the class **Cat** and more particularly at the two constructors of the class:

```
public class Cat
{
   // Field name
   private string name;
   // Field color
   private string color;

   …

   // Parameterless constructor
   public Cat()
   {
      this.name = "Unnamed";
      this.color = "gray";
   }

   // Constructor with parameters
   public Cat(string name, string color)
   {
      this.name = name;
      this.color = color;
   }

   …
}
```

We are going to use these constructors to illustrate the usage of constructors with and without parameters. For the class **Cat** defined that way we are going to give an example of creating its instances by each of the two constructors. One of the objects is going to be an ordinary undefined cat, and the other – our brown cat Johnny. After that we are going to execute the method **SayMiau** for each of the cats and analyze the result. Source code follows:

```
class CatManipulating
{
   static void Main()
   {
      Cat someCat = new Cat();

      someCat.SayMiau();
      Console.WriteLine("The color of cat {0} is {1}.",
        someCat.Name, someCat.Color);

      Cat someCat = new Cat("Johnny", "brown");

      someCat.SayMiau();
      Console.WriteLine("The color of cat {0} is {1}.",
        someCat.Name, someCat.Color);
    }
}
```

As a result of the program's execution the following text is printed on the standard output:

```
Cat Unnamed said: Miauuuuuu!
The color of cat Unnamed is gray.
Cat Johnny said: Miauuuuuu!
The color of cat Johnny is brown.
```

Static Fields and Methods

The data members, which we considered up until, now implement **states of the objects** and are directly related to specific instances of the classes. In OOP there are special categories fields and methods, which are associated with the data type (class), and not with the specific instance (object). We call them **static members** because are independent of concrete objects. Furthermore, they are used without the need of creating an instance of the class in which they are defined. They can be fields, methods and constructors. Let's consider shortly static members in C#.

A static field or method in a given class is defined with the keyword **`static`**, placed before the type of the field or the type of returned value of the method. When defining a **static constructor**, the word static is placed before the name of the constructor. Static constructors are not going to be discussed in this chapter – for now we are going to consider only static fields and methods (the more curious readers can look up in MSDN).

When to Use Static Fields and Methods?

To find the answers of this question we have to understand very well the difference between static and non-static members. We are going to consider into details what it is.

We have already explained the main difference between the two types of members. Let's interpret the **class as a category of objects**, and **the object as a representative of this category**. Then the static members reflect the state and the behavior of the category itself, and the non-static the state and the behavior of the separate representatives of the category.

Now we are going to pay special attention to the **initialization of static and non-static fields**. We already know that non-static fields are initialized with the call to the constructor of the class when creating an instance of it – either inside the body of the constructor, or outside. However, the initialization of static fields cannot be performed when the object of the class is created, because they can be used without a created instance of the class. It is important to know the following:

⚠	**Static fields are initialized when the data type (the class) is used for the first time, during the execution of the program.**

Now we shall see how to use static fields and methods in practice.

Static Fields and Methods – Example

The example, which we are going to give, solves the following simple problem: we need a method that every time returns a value greater with one than the value returned at the previous call of the method. We choose the first returned value to be 0. Obviously this method generates the sequence of natural number. Similar functionality is widely used in practice, for example, for uniform numbering of objects. Now we are going to see how this could be implemented with the means of OOP.

Let's assume that the method is called **NextValue()** and is defined in a class called **Sequence**. The class has a field **currentValue** from type **int**, which contains the last returned value by the method. We would like the following two actions to be performed consecutively in the method body: the value of the field to be increased and its new value to be returned as a result. Obviously the returned by the method value does not depend on the concrete instance of the class **Sequence**. For this reason the method and the field are static. You can now see the described implementation of the class:

```
public class Sequence
{
  // Static field, holding the current sequence value
  private static int currentValue = 0;

  // Intentionally deny instantiation of this class
  private Sequence()
  {
  }

  // Static method for taking the next sequence value
  public static int NextValue()
  {
    currentValue++;
    return currentValue;
```

```
   }
}
```

The observant reader has noticed that the so defined class has a default constructor, which is declared as **private**. This usage of a constructor may seem strange, but is quite deliberate. It is good to know the following:

A class that has only private constructors cannot be instantiated. Such class usually has only static members and is called "utility class".

For now we are not going to go into details about the **access modifiers public**, **private** and **protected**. We shall explain them comprehensively in the chapter "Defining Classes".

Let's take a look at a simple program, which uses the class **Sequence**:

```
class SequenceManipulating
{
   static void Main()
   {
      Console.WriteLine("Sequence[1...3]: {0}, {1}, {2}",
         Sequence.NextValue(), Sequence.NextValue(),
         Sequence.NextValue());
   }
}
```

The example prints on the standard output the first three natural numbers by triple consecutive call of the method **NextValue()** of the class **Sequence**. The result from this code is the following:

```
Sequence[1...3]: 1, 2, 3
```

If we try to create several different sequences, as the constructor of the class **Sequence** is declared **private**, we are going to get compile time error.

Examples of System C# Classes

After we got acquainted with the basic functionality of objects, we are going to consider briefly several **commonly used system classes** from the standard library of .NET Framework. This way we are going to see in practice the so far explained material, and also show how system classes ease our every-day work.

The System.Environment Class

We start with one of the basic system classes in .NET Framework: **System.Environment**. It contains a set of useful fields and methods, which ease getting information about the hardware and the operating system, and some of them, give the ability to interact with the program environment. Here is a part of the functionality provided by this class:

- Information about the processors count, the computer network name, the version of the operating system, the name of the current user, the current directory, etc.
- Access to externally defined properties and environment variables, which we are not going to consider in this book.

Now we are going to show one interesting application of a method of the class **Environment**, which is commonly used in practice when developing programs with critical fast performance. We are going to detect the time needed for the execution of the source code with the help of the property **TickCount**. Here it is how it works:

```
class SystemTest
{
  static void Main()
  {
    int sum = 0;
    int startTime = Environment.TickCount;

    // The code fragment to be tested
    for (int i = 0; i < 10000000; i++)
    {
      sum++;
    }

    int endTime = Environment.TickCount;
    Console.WriteLine("The time elapsed is {0} sec.",
      (endTime - startTime) / 1000.0);
  }
}
```

The static property **TickCount** of the class **Environment** returns as a result the count of milliseconds that have passed since the computer is on until the time of the method call. With its help we detect the milliseconds past before and after the execution of the source code. Their difference is the wanted time for the execution of the fragment source code measured in milliseconds.

As a result of the execution of the program on the standard output we print the result of the following type (the measured time varies according to the current computer configuration and its load):

```
The time elapsed is 0.031 sec.
```

In the example we have used two static members of two system classes: the static property **Environment.TickCount** and the static method **Console.WriteLine(…)**.

The System.String Class

We have already met the **String** (**System.String**) class of .NET Framework, which represents strings. Let's recall that we can think of strings as a primitive data type in C#, although the work with them is different from the work with different primitive data types (integers, floating point numbers, Boolean variables, etc.). We are going to describe them in details in the chapter "Strings and Text Processing".

The System.Math Class

The **System.Math** class contains methods for performing basic **numeric and mathematical operations** such as raising a number to a power, taking a logarithm and square root, and some trigonometric functions. We are going to give a simple example, which illustrates its usage.

We want to make a program, which calculates the area of a triangle by given two sides and an angle between them in degrees. Therefore we need the method **Sin(…)** and the constant **PI** of the class **Math**. With the help of the **π** number we can easily convert to radians the entered in degrees angle. You can see an example implementation of the described logic:

```
class MathTest
{
    static void Main()
    {
        Console.WriteLine("Length of the first side:");
        double a = double.Parse(Console.ReadLine());
        Console.WriteLine("Length of the second side:");
        double b = double.Parse(Console.ReadLine());
        Console.WriteLine("Size of the angle in degrees:");
        int angle = int.Parse(Console.ReadLine());

        double angleInRadians = Math.PI * angle / 180.0;
        Console.WriteLine("Area of the triangle: {0}",
            0.5 * a * b * Math.Sin(angleInRadians));
    }
}
```

We can easily test the program if we check whether it calculates correctly the **area of an equilateral triangle**. For further convenience we choose the length of the side to be 2 – then we find the area with the well-known formula:

$$S = \frac{\sqrt{3}}{4} 2^2 = \sqrt{3} = 1{,}7320508\ ...$$

We enter consecutively the numbers 2, 2, 60 and on the standard output we can see:

```
Face of the triangle: 1.73205080756888
```

Depending on your system localization (Region and Language Settings) your output might be "**1,73205080756888**" or "**1.73205080756888**". You might fix the decimal point to "**.**" by this line of code, executed at your program start:

```
System.Threading.Thread.CurrentThread.CurrentCulture =
  System.Globalization.CultureInfo.InvariantCulture;
```

The System.Math Class – More Examples

As we already saw, apart from mathematical methods, the **Math** class also defines two well known in mathematics constants: the trigonometric constant **π** and the Euler's number ***e***. Here is an example with them:

```
Console.WriteLine(Math.PI);
Console.WriteLine(Math.E);
```

When executing the code above, we get the following output:

```
3.141592653589793
2.718281828459045
```

The System.Random Class

Sometimes in programming we have to use **random numbers**. For instance, we would like to generate 6 random numbers in the range 1 to 49 (not necessarily unequal). This could be done by using the **System.Random** class and its method **Next()**. Before we use the **Random** class we have to create instance of it, at which point it is initialized with a random value (derived from the current system time in the operating system). After that we can randomly generate a number in the range **[0…n)** by calling the method **Next(n)**. Notice that this method can return zero, but always returns a random number smaller than the set value **n**. Therefore, if we would like to get a number in the range **[1…49]**, we have to use the expression **Next(49) + 1**.

Below is an example source code of a program, which generates 6 random numbers in the range from 1 to 49 by using the **Random** class (note that it is not guaranteed that the numbers are unique like in the classical Bulgarian lottery TOTO 6/49):

```
class RandomNumbersBetween1And49
{
    static void Main()
    {
        Random rand = new Random();
        for (int number = 1; number <= 6; number++)
        {
            int randomNumber = rand.Next(49) + 1;
            Console.Write("{0} ", randomNumber);
        }
    }
}
```

Here is how a possible output of the program looks like:

```
16 49 7 29 1 28
```

The System.Random Class – Generating a Random Password

To show you how useful **the random numbers generator** in .NET Framework can be, we are going to set as a task to **generate a random password** which is between 8 and 15 characters long, contains at least two capital letters, at least two small letters, at least one digit and at least three special chars. For this purpose we are going to use the following algorithm:

1. We start with an empty password. We create a generator of random numbers.
2. We generate twice a random capital letter and place it at a random position in the password.
3. We generate twice a random small letter and place it at a random position in the password.
4. We generate twice a random digit and place it at a random position in the password.
5. We generate three times a random special character and place it at a random position in the password.
6. Until this moment the password should consist of 8 characters. In order to supplement it to 15 characters at most, we can insert random count of times (between 0 and 7) at a random position in the password a random character (a capital letter, a small letter or a special char).

An implementation of the described algorithm is given below:

```
class RandomPasswordGenerator
{
    private const string CapitalLetters =
        "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    private const string SmallLetters =
        "abcdefghijklmnopqrstuvwxyz";
    private const string Digits = "0123456789";
    private const string SpecialChars =
        "~!@#$%^&*()_+=`{}[]\\|':;.,/?<>";
    private const string AllChars =
        CapitalLetters + SmallLetters + Digits + SpecialChars;

    private static Random rnd = new Random();

    static void Main()
    {
        StringBuilder password = new StringBuilder();

        // Generate two random capital letters
        for (int i = 1; i <= 2; i++)
        {
            char capitalLetter = GenerateChar(CapitalLetters);
            InsertAtRandomPosition(password, capitalLetter);
        }

        // Generate two random small letters
        for (int i = 1; i <= 2; i++)
        {
            char smallLetter = GenerateChar(SmallLetters);
            InsertAtRandomPosition(password, smallLetter);
        }

        // Generate one random digit
        char digit = GenerateChar(Digits);
        InsertAtRandomPosition(password, digit);

        // Generate 3 special characters
        for (int i = 1; i <= 3; i++)
        {
            char specialChar = GenerateChar(SpecialChars);
            InsertAtRandomPosition(password, specialChar);
        }

        // Generate few random characters (between 0 and 7)
        int count = rnd.Next(8);
        for (int i = 1; i <= count; i++)
        {
            char specialChar = GenerateChar(AllChars);
            InsertAtRandomPosition(password, specialChar);
        }

        Console.WriteLine(password);
    }

    private static void InsertAtRandomPosition(
        StringBuilder password, char character)
    {
        int randomPosition = rnd.Next(password.Length + 1);
        password.Insert(randomPosition, character);
    }

    private static char GenerateChar(string availableChars)
    {
        int randomIndex = rnd.Next(availableChars.Length);
        char randomChar = availableChars[randomIndex];
        return randomChar;
    }
}
```

Let's explain several unclear moments in the source code. Let's start from the definition of the constants:

```
private const string CapitalLetters =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
private const string SmallLetters =
    "abcdefghijklmnopqrstuvwxyz";
private const string Digits = "0123456789";
private const string SpecialChars =
    "~!@#$%^&*()_+=`{}[]\\|':;.,/?<>";
private const string AllChars =
    CapitalLetters + SmallLetters + Digits + SpecialChars;
```

Constants in C# are immutable variables whose values are assigned during their initialization in the source code of the program and after that they cannot be changed. They are declared with the modifier **const**. They are used for defining a number or a string, which afterwards is used many times in the program. This way repetition of certain values in the code is avoided and these values can be easily altered by changing only one place in the code. For example, if in a certain moment we decide that the character "**,**" (comma) should not be used when generating a password, we can change only one row in the program (the corresponding constant) and the change is going to reflect on every row where the constant is being used. In C# constants are written in Pascal Case (the words in the name, merged together, each of them starts with an uppercase letter, and the rest of them are lowercase). More about constants we will learn in the section "Constants" in the chapter "Defining Classes".

Let's explain how the other parts of the program work. In the beginning, as a static member variable in the class **RandomPasswordGenerator** is created the random number generator **rnd**. As this variable **rnd** is defined in the class (not in the **Main()** method), it is accessible by the whole class (by each of its methods), and as it is defined static, it is accessible by the static methods, too. Thus, anywhere the program needs a random integer variable the same random number generator is used. It is initialized when the class **RandomPasswordGenerator** is loaded.

The method **GenerateChar()** returns a randomly chosen character in a set of characters given as a parameter. It works very simply: it chooses a random position in the set of characters (between 0 and the count of characters minus 1) and returns the characters at this position.

The method **InsertAtRandomPosition()** is not complicated too. It chooses a random position in the **StringBuilder** object, which is passed and inserts on this position the returned character. We are going to pay special attention to the class **StringBuilder** in the chapter "Strings and Text Processing".

Here is a sample output of the program for generating passwords, which we just considered (this output is different at each program run due to its randomness by nature):

```
8p#Rv*yTl{tN4
```

Namespaces

Namespace (package) in OOP we call a **container for a group of classes**, which are united by a common feature or are used in a common context. The namespaces contribute to a better logical organization of the source code by creating a semantic division of the classes in categories and makes easier their usage in the source code. Now we are going to consider namespaces in C# and are going to see how we can use them.

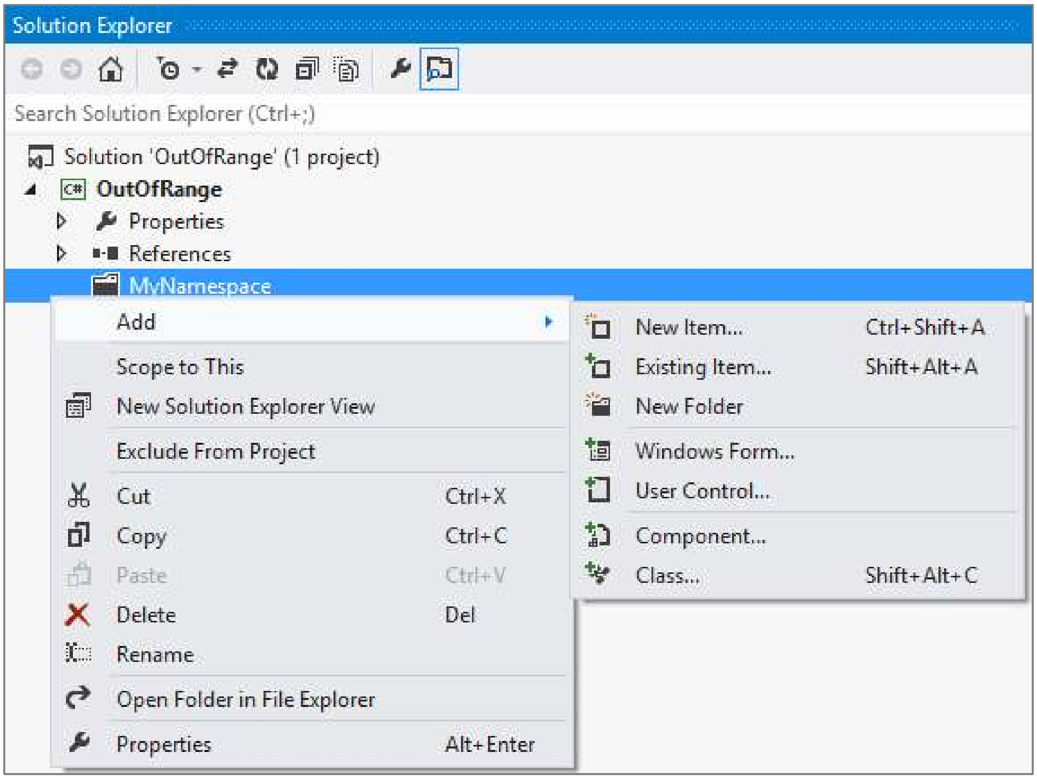
What Are Namespaces in C#?

Namespaces in C# are **named groups of classes**, which are logically related without any specific requirement on how to be placed in the file system. However, it is considered that the folder name should match the namespace name and the names of the files should match the names of the classes, which are defined in them. We have to note that in some programming languages the compilation of the source code in a given namespace depends on the distribution of the elements of the namespace in folders and files on the disk. In Java, for instance, the described file organization is mandatory (if it is not followed, compilation errors occur). C# is not so strict regarding this.

Now, let’s consider the mechanism for defining namespaces.

Defining Namespaces

In case we like to create a new namespace or a new class which belongs to a given namespace, in Visual Studio this happens automatically by the commands in the context menu of the Solution Explorer (on right click on the corresponding folder). By default the Solution Explorer is visualized like a Dock in the right part of the integrated environment. We are going to illustrate how we could add a new class in the already existing namespace **`MyNamespace`** by the context menu of Solution Explorer in Visual Studio:



As the project is called **`MyConsoleApplication`** and we are adding in its folder **`MyNamespace`**, the newly created class is going to be in the following namespace:

```
namespace MyConsoleApplication.MyNamespace
```

If we have defined a class in its own file and we like to add it in a new or already existing namespace, it is not hard to do it manually. It is enough to change the named block with a keyword **`namespace`** in the class:

```
namespace <namespace_name>
{
  …
}
```

In the definition we use the keyword **namespace**, followed by the full name of the namespace. It is considered that the namespaces in C# start with a capital letter and are written in Pascal Case. For example, if we have to make a namespace containing classes for string processing, it is desirable we name it **StringUtils**, and not **string_utils**.

Nested Namespaces

Except classes, **namespaces can contain other namespaces** in themselves (nested namespaces). This way, intuitively we create a hierarchy of namespaces, which allows even more precise distribution of classes according to their semantics.

When naming namespaces in the hierarchy we use the character **.** as a separator (dot notation). For example, the namespace **System** from .NET Framework contains in itself the sub-namespace **Collections** and thus the full name of the nested namespace **Collections** is **System.Collections**.

Full Names of Classes

In order to absolutely understand the meaning of namespaces, it is important for us to know the following:

Classes are required to have unique names only within the namespaces, in which they are defined.

Outside a given namespace we can have classes with random names regardless of whether they match with any of the names of classes in the namespace. This is because classes in the namespace are uniquely defined in its context. It is time to see how to define syntactically this uniqueness.

Full name of the class we call the first name of the class, preceded by the name of the namespace in which it is defined. The full name of each class is unique. Again we use dot notation:

```
<namespace_name>.<class_name>
```

Let's take, for example, the system class **CultureInfo**, defined in the namespace **System.Globalization** (we have already used it in the chapter "Console Input and Output"). According to the definition, the full name of the class is **System.Globalization.CultureInfo**.

In .NET Framework sometimes there are classes from different namespaces with matching names, for example:

```
System.Windows.Forms.Control
System.Web.UI.Control
System.Windows.Controls.Control
```

Inclusion of a Namespace

When building an application according to the object area, very often it is necessary to use the classes of a namespace multiple times. For the programmer's convenience there is a mechanism for **inclusion of a namespace** in the current file with a source code. After the given namespace is included, all classes defined in it may be used without the need to use their full names.

The inclusion of a namespace in the current source code file is executed with the **keyword using** in the following way:

```
using <namespace_name>;
```

We are going to pay attention to an important feature of including namespaces in the described way. All classes defined directly in the namespace **<namespace_name>** are included and can be used, but we have to know the following:

Inclusion of namespaces is not recursive, i.e. when including a namespace the classes from the nested namespaces are not included.

For example, the inclusion of namespaces **System.Collections does not** automatically include the classes from its nested namespace **System.Collections.Generic**. When used, either we have to apply their full names, or to include the namespace, which contains them.

Using a Namespace – Example

In order to illustrate the principle of inclusion of a namespace, we are going to consider the following program which reads numbers, saves them in lists and counts how many of them are integer numbers and how many are double:

```
class NamespaceImportTest
{
  static void Main()
  {
    System.Collections.Generic.List<int> ints =
      new System.Collections.Generic.List<int>();
    System.Collections.Generic.List<double> doubles =
      new System.Collections.Generic.List<double>();

    while (true)
    {
      int intResult;
      double doubleResult;
      Console.WriteLine("Enter an int or a double:");
      string input = Console.ReadLine();

      if (int.TryParse(input, out intResult))
      {
        ints.Add(intResult);
      }
      else if (double.TryParse(input, out doubleResult))
      {
        doubles.Add(doubleResult);
      }
      else
      {
        break;
      }
    }

    Console.Write("You entered {0} ints:", ints.Count);
    foreach (var i in ints)
    {
      Console.Write(" " + i);
    }
    Console.WriteLine();

    Console.Write("You entered {0} doubles:", doubles.Count);
    foreach (var d in doubles)
    {
      Console.Write(" " + d);
    }
    Console.WriteLine();
  }
}
```

For this purpose the program uses the class **`System.Collections.Generic.List`** as it calls it by its full name.

Let's see how the program above works: we enter consecutively the values **4**, **1.53**, **0.26**, **7**, **2**, **end**. We get the following result on the standard output:

```
You entered 3 ints: 4 7 2
You entered 2 doubles: 1.53 0.26
```

The program does the following: it gives the user the opportunity to enter consecutively numbers, which may be integer or double. This continues until the moment in which a value different from a number is entered. Then on the standard output two rows are displayed, respectively with integer and double numbers.

For the implementation of the described actions we use two helping objects respectively of type **System.Collections.Generic.List<int>** and **System.Collections.Generic.List<double>**. Obviously, the full names of the classes make the code unreadable, and cause inconveniences. We can easily avoid this effect by including the namespace **System.Collections.Generic** and use directly the classes by name. You can now see the shortened version of the program above:

```
using System.Collections.Generic;

class NamespaceImportTest
{
    static void Main()
    {
        List<int> ints = new List<int>();
        List<double> doubles = new List<double>();
        …
    }
}
```

Exercises

1. Write a program, which reads from the console a year and **checks if it is a leap year**.
2. Write a program, which generates and prints on the console **10 random numbers** in the range [100, 200].
3. Write a program, which prints, on the console **which day of the week is today**.
4. Write a program, which prints on the standard output the **count of days, hours, and minutes, which have passes since the computer is started** until the moment of the program execution. For the implementation use the class **Environment**.
5. Write a program which by given two sides **finds the hypotenuse of a right triangle**. Implement entering of the lengths of the sides from the

standard input, and for the calculation of the hypotenuse use methods of the class **Math**.

6. Write a program which **calculates the area of a triangle** with the following given:
   - three sides;
   - side and the altitude to it;
   - two sides and the angle between them in degrees.
7. Define your own namespace **CreatingAndUsingObjects** and place in it two classes **Cat** and **Sequence**, which we used in the examples of the current chapter. Define one more namespace and make a class, which calls the classes **Cat** and **Sequence**, in it.
8. Write a program which creates 10 objects of type **Cat**, gives them names **CatN**, where **N** is a unique serial number of the object, and in the end call the method **SayMiau()** for each of them. For the implementation use the namespace **CreatingAndUsingObjects**.
9. Write a program, which **calculates the count of workdays between the current date and another given date** after the current (inclusive). Consider that workdays are all days from Monday to Friday, which are not public holidays, except when Saturday is a working day. The program should keep a list of predefined public holidays, as well as a list of predefined working Saturdays.
10. You are given a **sequence of positive integer numbers** given as string of numbers separated by a space. Write a program, which **calculates their sum**. Example: "**43 68 9 23 318**" → **461**.
11. Write a program, which **generates a random advertising message** for some product. The message has to consist of laudatory phrase, followed by a laudatory story, followed by author (first and last name) and city, which are selected from predefined lists. For example, let's have the following lists:
    - **Laudatory phrases**: {"The product is excellent.", "This is a great product.", "I use this product constantly.", "This is the best product from this category."}.
    - **Laudatory stories**: {"Now I feel better.", "I managed to change.", "It made some miracle.", "I can't believe it, but now I am feeling great.", "You should try it, too. I am very satisfied."}.
    - **First name** of the author: {"Dayan", "Stella", "Hellen", "Kate"}.
    - **Last name** of the author: {"Johnson", "Peterson", "Charls"}.
    - **Cities**: {"London", "Paris", "Berlin", "New York", "Madrid"}.

    Then the program would print randomly generated advertising message like the following:

```
I use this product constantly. You should try it, too. I am
very satisfied. -- Hellen Peterson, Berlin
```

12. * Write a program, which calculates the value of a given numeral expression given as a string. The numeral expression consists of:
    - real numbers, for example **5**, **18.33**, **3.14159**, **12.6**;
    - arithmetic operations: **+**, **-**, *****, **/** (with their standard priorities);
    - mathematical functions: **ln(x)**, **sqrt(x)**, **pow(x, y)**;
    - brackets for changing the priorities of the operations: **(** and **)**.

Note that the numeral expressions have priorities, for example the expression **-1 + 2 + 3 * 4 - 0.5 = (-1) + 2 + (3 * 4) - 0.5 = 12.5**.

Solutions and Guidelines

1. Use **DateTime.IsLeapYear(year)**.
2. Use the class **Random**. You may generate random numbers in the range [100, 200] by calling **Random.Next(100, 201)**.
3. Use **DateTime.Today.DayOfWeek**.
4. Use the property **Environment.TickCount**, in order to get the count of passed milliseconds. Use the fact that one second has 1,000 milliseconds; one minute has 60 seconds; one hour has 60 minutes and one day has 24 hours.
5. The hypotenuse of a rectangular triangle could be found with the **Pythagorean Theorem** $\mathbf{a}^2 + \mathbf{b}^2 = \mathbf{c}^2$, where **a** and **b** are the two sides, and **c** is the hypotenuse. Take square root of the two sides of the equation in order to get the length of the hypotenuse. Use the **Sqrt(…)** methods of the **Math** class.
6. For the first sub-problem of the task use the **Heron's Formula** $S = \sqrt{p(p-a)(p-b)(p-c)}$, where $p = \frac{a+b+c}{2}$. For the second sub-problem use the **formula**: $S = \frac{a*h_a}{2}$. For the third sub-problem use the **formula**: $S = \frac{a*b*sin(\gamma)}{2}$. For the sine use the **System.Math** class.
7. Make a **new project in Visual Studio**, right click on the folder and choose the menu **Add → New Folder**. Then enter the name of the folder and press [Enter], right click on the newly made folder and choose **Add → New Item…** from the list choose **Class**, for the name of the new class enter **Cat** and press [Add]. Change the definition of the newly created class with the definition, which we gave to this chapter, to put the classes in a **namespace**. Make the same to the class **Sequence**.

8. Create an array with 10 elements of type **Cat**. Create 10 objects of type **Cat** in a loop (use a constructor with parameters) and assign them to the corresponding element of the array. For the serial number of the objects use the method **NextValue()** of the **Sequence** class. In the end again in an array use the method **SayMiau()** for each of the array elements.

9. Use the class **System.DateTime** and the methods in it. You can execute a loop from the current date (**DateTime.Now.Date**) to the end date, consecutively incrementing the day by the method **AddDays(1)** and count the working days according to your country (e.g. all days except Saturday and Sunday and a few fixed non-working official holidays).

   Another approach that might work is to **subtract the dates** to find the **TimeSpan** between them (**DateTime** values can be subtracted, just like a numbers). This will give you the count of days between the dates. You will need to perform some additional calculations to find how much weekends are included in this count and discard them.

10. Use **String.Split(' ')** to split the string by spaces. Then use **Int32.Parse(…)** to extract the separate numbers from the obtained **string** array as **int** values and sum them.

11. Use the class **System.Random** and its method **Next(…)** to select a random laudatory phrase, laudatory story, first name, last name and city and combine them.

12. **Calculating a numeral expression** is **quite hard** and is unlikely a beginner programmer to solve it correctly without external help. As a start check out the article in Wikipedia about the "**Shunting-yard algorithm**" (en.wikipedia.org/wiki/Shunting-yard_algorithm) describing how to convert an expression from to **postfix notation** (reversed Polish notation), and the article about **calculating a postfix expression** (en.wikipedia.org/wiki/Reverse_Polish_notation). There are really much special cases, so be sure to test your solution carefully.