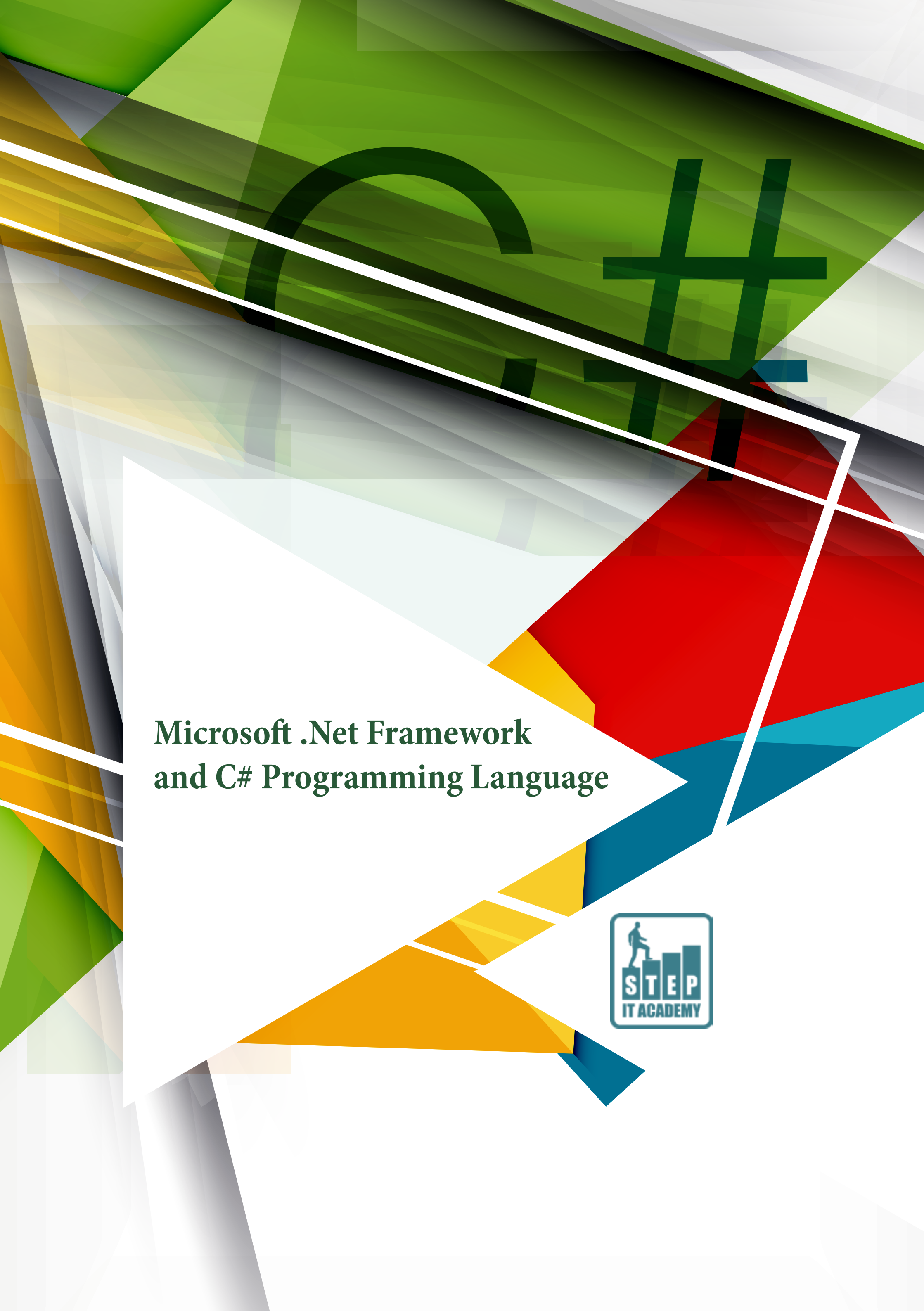


Microsoft .Net Framework and C# Programming Language

STEP IT ACADEMY

lesson 3

Exception handling and namespaces, operator overloading, indexers and properties

Contents

1. Structs

The concept of struct

Struct is a composite data type that is a sequence of variables of different types. The .NET structs of C# are designed for grouping and storing small amounts of data.

Structs are value types inherited from `System.ValueType`. This means that they are either stored on the stack, or they are embedded (if they are a part of another object stored in the heap), and have the same lifetime limitations as for simple data types.

The syntax of struct declaration

The `struct` keyword is used for struct declaration:

```
struct Имя : Интерфейсы
{
   //members declaration
}
```

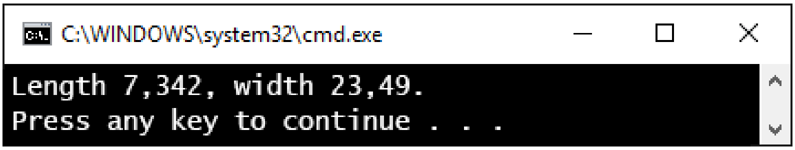
Here is an example of working with the struct. Let's write a program that will work with the struct that describes some dimensions.

```
struct Dimensions
{
   public double Length;
   public double Width;
}
```

It is possible to define the fields, properties, methods, constructors, iterators and events (some of which you will learn later) in the structs. For example:

```
struct Dimensions
{
   private double Length;
   public double Width;

   public Dimensions(double length, double width)
   {
      Length = length;
      Width = width;
   }

   public void Print()
   {
      Console.WriteLine($"Length {Length},
                              width {Width}.");
   }
}

class Program
{
   static void Main(string[] args)
   {
      double length = 7.342, width = 23.49;
      Dimensions dimensions =
                  new Dimensions(length, width);
      dimensions.Print();
   }
}
```

The result of the program execution (Figure 1.1):



```
C:\WINDOWS\system32\cmd.exe
Length 7,342, width 23,49.
Press any key to continue . . .
```

Figure 1.1. Example of creating a struct

The need and features of structs application

Structs are used for performance optimization, when the only class function needed is the storage of some small chunks of data. The structs can be interpreted as simplified classes.

The `struct` type does not require an individual reference variable. This means that the use of structs consumes less memory. Due to the direct access to the structs, there is no loss in performance when working with them, which takes place when accessing the class objects. The memory for the struct is allocated and removed faster due to storing it on the stack.

When assigning a struct to another one, or when passing a struct to a method as a parameter, the entire contents of a struct are copied, that is much slower than passing a reference. (Copying speed depends on the size of the struct: the more fields there are, the more time it takes to copy).

The structs have access to the `System.Object` class methods, which can override. The structs do not support inheritance. The structs can inherit interfaces.

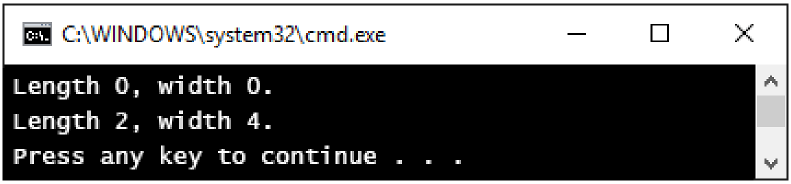
Constructor without parameters and struct

The compiler always generates a default constructor without parameters, which cannot be overridden. The memory allocation for the entire struct occurs when declaring a variable. Accordingly, the new operator acts differently for structs than the reference types: it will cause the initialization of fields with default values.

```
public void DoSomething()
{
   Dimensions d;
   //memory is allocated, but the field
   //are not initialized with values;
```

```
    d = new Dimensions();
       d.Print(); //Length equals 0, Width equals 0;
    d.Length += 2;
    d.Width += 4;
    d.Print();   //Length equals 2, Width equals 4;
}
```

Code outcome is shown in the Figure 1.2:

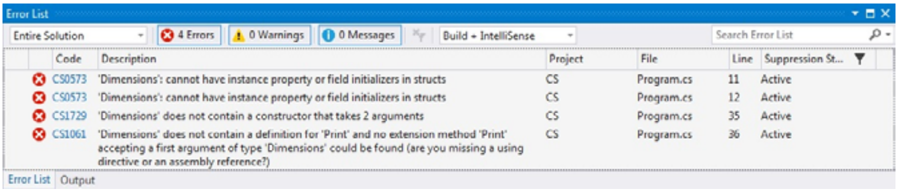


```
C:\WINDOWS\system32\cmd.exe
Length 0, width 0.
Length 2, width 4.
Press any key to continue . . .
```

Figure 1.2. Example of working with the struct

The default constructor initializes all the fields with zero values. It is also impossible to ignore the default constructor by specifying the initial values of the fields. The following code will cause a compilation error (Figure 1.3):

```
struct Dimensions
{
    public double Length = 3;
    public double Width = 4;
}
```



Error List

Entire Solution | 4 Errors | 0 Warnings | 0 Messages | Build + IntelliSense | Search Error List

Code	Description	Project	File	Line	Suppression St...
CS0573	'Dimensions': cannot have instance property or field initializers in structs	CS	Program.cs	11	Active
CS0573	'Dimensions': cannot have instance property or field initializers in structs	CS	Program.cs	12	Active
CS1729	'Dimensions' does not contain a constructor that takes 2 arguments	CS	Program.cs	35	Active
CS1061	'Dimensions' does not contain a definition for 'Print' and no extension method 'Print' accepting a first argument of type 'Dimensions' could be found (are you missing a using directive or an assembly reference?)	CS	Program.cs	36	Active

Error List | Output

Figure 1.3. An error occurred while initializing the fields of the struct

2. The syntax of class declaration

The C# classes are templates, using which you can create objects. Each object contains data and methods that manipulate this data. The class defines data and functionality that each particular object (sometimes called an instance) of the class can have. For example, if you have a class that represents a student, it can define the fields like _studentID, _firstName, _lastName, _group, and others, which it needs for storing information on a particular student.

The class can also define the functionality that works with the data stored in these fields. You create an instance of this class to represent a particular student, set the values of the instance fields and use its functionality. The new keyword is used for allocating memory for the instance when creating classes, just as all the reference types. As a result, the object will be created and initialized (remember that the numeric fields are initialized with zeros, the logical fields are initialized with false, and the reference — with null).

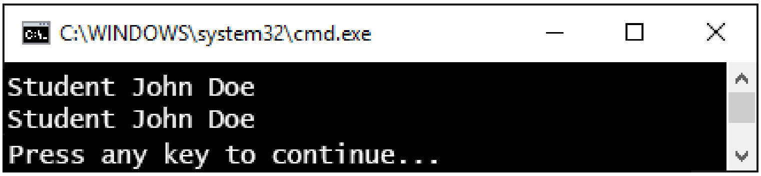
The syntax for class declaration and initialization is as follows:

```
[modifiers] Class class_name
{
    [modifiers] data_type field_name;
    [modifiers] data_type field_name;
    ...
    [modifiers] data_type field_name([parameters])
```

```
        {
        // method body
        }
        ...
}
```

The example of declaring a class that describes a student (Figure 2.1):

```
class Student
{
    int _studentID;
    string _firstName = "John";
    string _lastName = "Doe";
    string _group;

    public void Print()
    {
        Console.WriteLine($"Student {_firstName }
                            {_lastName}");
    }
}

class Program
{

    static void Main(string[] args)
    {
        Student st1;
        st1 = new Student();
        st1.Print();

        Student st2 = new Student();
        st2.Print();

    }
}
```


C:\WINDOWS\system32\cmd.exe

```
Student John Doe
Student John Doe
Press any key to continue...
```

Figure 2.1. Example of class declaration

The access to the fields and methods of the class is performed using the "." operator, which is called from the class object. It should also be remembered that we can access only the public data of the class beyond its bounds. The rest of the data remains encapsulated. We will now study the modifiers.

3. Access modifiers of the C# programming language

All the types and their members have a level of accessibility, which determines the possibility to use them in other code of the developer's assembly, or in other assemblies.

When defining a class, it can be made public or internal. `Public` type is accessible by any code of any assembly. `Internal` class is available only in the assembly, wherein it is defined. By default, the C# compiler makes the class internal.

When determining a class member (including nested one), you can specify the access modifier to this member. Modifiers determine what members can be referenced from the code. The CLR has its set of possible access modifiers, but each programming language has its own syntax and terms. Let's consider the modifiers that define the level of restriction: from maximum (`private`) to minimum (`public`):

- **private** — data is accessible only by the methods within the class and the nested classes;
- **protected** — data is accessible only by the methods within the class (and nested classes), or by any of its child classes;
- **internal** — data is accessible only by the methods of the current assembly;
- **protected internal** — data is accessible only by the methods of nested or derived class type and by any methods of the current assembly;

- **public** — data is accessible by all the methods in all the ssemblies.

You should also understand that the class member can be accessed only if it is defined in the visible class. In other words, if the internal class that has a public method is defined in the assembly A, then the code of assembly B will not be able to call the public method, because the internal class of the assembly A cannot be accessed by the assembly B.

When compiling the code, the compiler verifies the code reference to classes and members. A compilation error occurs if there is an incorrect reference to any of the classes or members.

If you the access modifier is not explicitly defined, the C# compiler will choose `private` for the class members and `internal` for the class itself by default.

If a member of a base class is overridden in a derived class, then the C# compiler will require that the members of the base and derived classes have the same access modifier. The CLR allows lowering but not raising the level of access to the member when inheriting to the base class.

4. Class Fields

Any class can include a set of data. This data includes fields, class constructors, methods, overloaded operators, properties, events and nested classes. Let us consider the fields first.

A field is a variable that stores the value of any standard type or a reference to a reference type. The following keywords can be specified when declaring fields:

- **Static** is used for declaring a static field, which belongs to the class, rather than a specific object, i.e. it is common to all instances of the class.
- **Const** is used for declaring a constant field, i.e. the value of this field cannot be changed. The field with the const modifier must always be initialized when declaring the class field. Constant fields are implicitly static, so must be accessed only by the type name.
- **Readonly** indicates that the field will be used for reading only, the values can be assigned to such fields only in a constructor or immediately at declaration.

The CLR supports mutable (`read/write`) and immutable (`readonly`) fields. Most of the fields are mutable. This means that the value of such fields can be changed multiple times during code execution. You have already seen such fields in the previous example in the `Student` class. Immutable fields are more flexible than the mutable, because the value can be set to them dynamically, this value won't change at runtime or after it. It is important to understand that the immutability of the reference type field means the immutability of the reference it contains, but not the object which it references to. In other words, we cannot redirect a reference to another location in memory, but we can change the value of the object which it references to. We cannot change the values of the immutable fields of the value types.

Consider the following example:

```
class MyClass
{
   public readonly int var = 10;
   public readonly int[] myArr = { 1, 2, 3 };
}

class Program
{
   static void Main(string[] args)
   {
      MyClass obj = new MyClass();
      obj.var = 100;                //Error
      obj.myArr = new int[10];      //Error
      obj.myArr[0] = 11;            //No error
   }
}
```

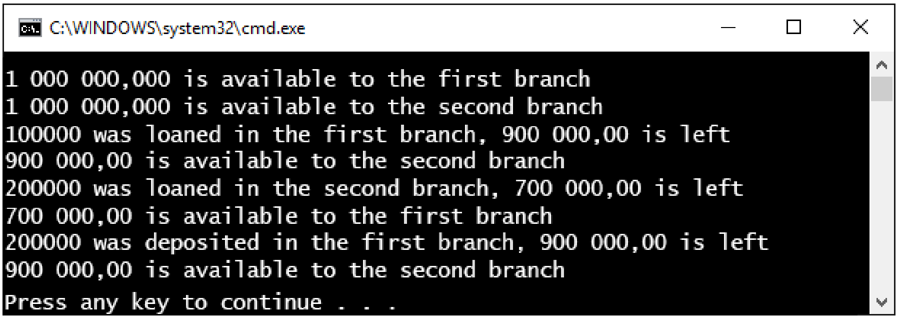
If a static field is declared, then it belongs to the class in general rather than to a specific object. And, accordingly, this object can be accessed only by the class name using the following syntax:

```
class_name.field_name
```

Static class field is common to all the instances of this class. For example, let us have the Bank class. This class has a balance static field. Let's simulate a situation, in which it is possible to put money on deposit or take out a loan at any branch of the bank. Let all the branches work with a joint account.

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

Console.WriteLine("{0:C} is available to the first
branch", Bank.balance);

Console.WriteLine("{0:C} is available to the second
branch", Bank.balance);

Console.WriteLine("100000 was loaned in the first
branch" + ", {0:C} is left", Bank.balance-= 100000);

Console.WriteLine("{0:C} is available to the second
branch", Bank.balance);

Console.WriteLine ("200000 was loaned in the second
branch " + ",{0:C} is left", Bank.balance -= 200000);

Console.WriteLine("{0:C} is available to the first
branch", Bank.balance);

Console.WriteLine("200000 was deposited in the
first branch" + ", {0: C} is left", Bank.balance +=
200000);

Console.WriteLine("{0:C} is available to the second
branch", Bank.balance);
    }
}
```

Program outcome (Figure 4.1):



```
1 000 000,000 is available to the first branch
1 000 000,000 is available to the second branch
100000 was loaned in the first branch, 900 000,00 is left
900 000,00 is available to the second branch
200000 was loaned in the second branch, 700 000,00 is left
700 000,00 is available to the first branch
200000 was deposited in the first branch, 900 000,00 is left
900 000,00 is available to the second branch
Press any key to continue . . .
```

Figure 4.1. Example of using a static field of the class

5. Constructors

The concept of constructor

Constructor is a special class method that is called implicitly when creating an object using the `new` keyword.

There are three types of constructors in C#. Despite such diversity, the idea of each one is to perform certain actions when creating objects. Let's consider them one by one.

Default constructor does not take parameters and is provided by the compiler when creating a class. This constructor is useful, because it sets zeros to all numeric types, sets `false` to the logic types, and sets `null` to the reference types. Therefore, all the fields of the object will be initialized with default values. If necessary, you can override the default constructor for your needs.

Constructor with parameters can take the required number of parameters for initialization of the class fields with the specific values.

Static constructor belongs to the class, but not to the object. It is used to initialize static fields of the class. It is defined without any access modifier with the `static` keyword.

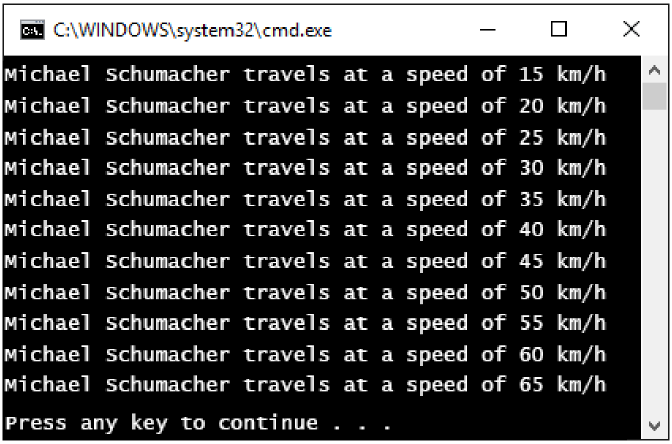
When creating constructors, it should be remembered that all the constructors (except static ones) must have a `public` access modifier, the same name as the class name, and must not return anything (even `void`). All the constructors but the default and static can have the required number of parameters. The default constructor can be only one.

Another important point: if you take on defining any constructor, then the default constructor provided to you by the compiler will not work!!! The compiler decides that you are now responsible for the creation of class objects, and you have no need in its "services", so it will not create the default constructor.

Let's consider an example of determining own default constructor, and the rest will be discussed in the following sections. For working with the constructors, let's create a new class that describes a car.

```
class Car
{
   private string _driverName; //Driver's name
   private int _currSpeed = 10;//Current speed

   public Car()                //Default constructor
   {
     driverName = "Michael Schumacher";
   }

   public void PrintState()    //Printing the current
                               //data
   {
     Console.WriteLine($"{_driverName}
                  travels at a speed of
                  {_currSpeed} km/h");
   }

   public void SpeedUp(int delta) //Speed increase
   {
      _currSpeed += delta;
   }
}

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

Program outcome (Figure 5.1):



C:\WINDOWS\system32\cmd.exe

```
Michael Schumacher travels at a speed of 15 km/h
Michael Schumacher travels at a speed of 20 km/h
Michael Schumacher travels at a speed of 25 km/h
Michael Schumacher travels at a speed of 30 km/h
Michael Schumacher travels at a speed of 35 km/h
Michael Schumacher travels at a speed of 40 km/h
Michael Schumacher travels at a speed of 45 km/h
Michael Schumacher travels at a speed of 50 km/h
Michael Schumacher travels at a speed of 55 km/h
Michael Schumacher travels at a speed of 60 km/h
Michael Schumacher travels at a speed of 65 km/h
Press any key to continue . . .
```

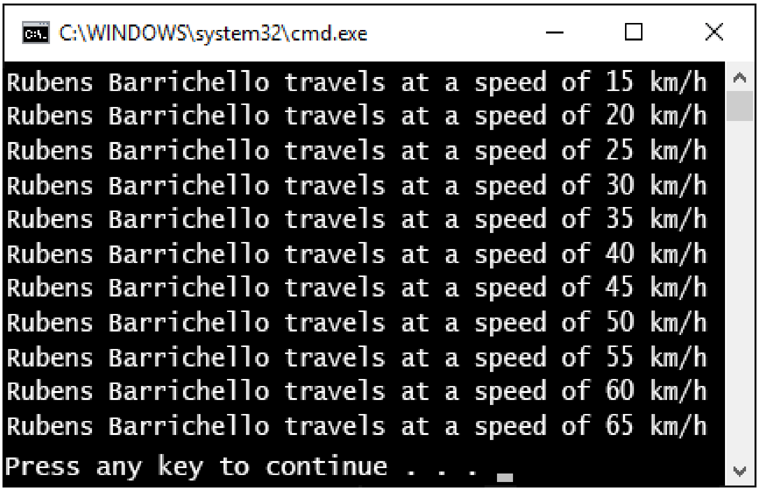
Figure 5.1. Use of the default constructor

Parameterized constructor

As mentioned above, the constructor with parameters differs from the default constructor by the presence of parameters. So we come straight to the example and add a parameterized constructor to our `Car` class. To save space, there are no previous fields and methods in the example, but they are in the full version.

```
class Car
{
//Previous fields and methods ...
   public Car(string name)
   {
        _driverName = name;
        _currSpeed = 10;
    }
}

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

Code outcome (Figure 5.2):



```
C:\WINDOWS\system32\cmd.exe
Rubens Barrichello travels at a speed of 15 km/h
Rubens Barrichello travels at a speed of 20 km/h
Rubens Barrichello travels at a speed of 25 km/h
Rubens Barrichello travels at a speed of 30 km/h
Rubens Barrichello travels at a speed of 35 km/h
Rubens Barrichello travels at a speed of 40 km/h
Rubens Barrichello travels at a speed of 45 km/h
Rubens Barrichello travels at a speed of 50 km/h
Rubens Barrichello travels at a speed of 55 km/h
Rubens Barrichello travels at a speed of 60 km/h
Rubens Barrichello travels at a speed of 65 km/h
Press any key to continue . . .
```

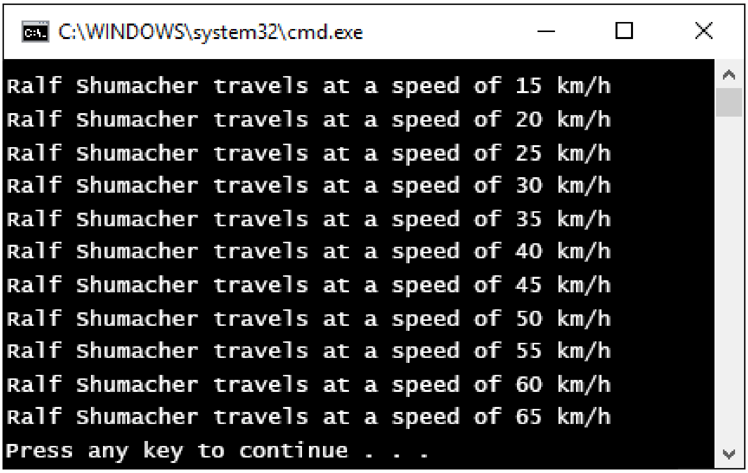
Figure 5.2. The use of the constructor with parameters

Overloaded constructors

Since the constructors are methods and, as we already know, the methods can be overloaded, then, consequently, there can be as much constructors as necessary. The only limitation: there should be only one default constructor. The number of parameterized constructors is restricted only by a common sense. Generally, try to define only the constructors, with the help of which it would be convenient to create an object. Let's go back to the car class and define another constructor.

```
class Car
{
   //Previous fields and methods ...
   public Car(string name, int speed)
   {
      _driverName = name;
       _currSpeed = speed;
   }
}

class Program
{
   static void Main(string[] args)
   {
      Car myCar = new Car("Ralf Shumacher", 15);
      for (int i = 0; i <= 10; i++)
      {
         myCar.SpeedUp(5);
         myCar.PrintState();
      }
   }
}
```

Code outcome (Figure 5.2):


C:\WINDOWS\system32\cmd.exe

```
Ralf Shumacher travels at a speed of 15 km/h
Ralf Shumacher travels at a speed of 20 km/h
Ralf Shumacher travels at a speed of 25 km/h
Ralf Shumacher travels at a speed of 30 km/h
Ralf Shumacher travels at a speed of 35 km/h
Ralf Shumacher travels at a speed of 40 km/h
Ralf Shumacher travels at a speed of 45 km/h
Ralf Shumacher travels at a speed of 50 km/h
Ralf Shumacher travels at a speed of 55 km/h
Ralf Shumacher travels at a speed of 60 km/h
Ralf Shumacher travels at a speed of 65 km/h
Press any key to continue . . .
```

Figure 5.3. The use of the constructor with parameters

Static constructor

Static constructor is linked with the class, rather than with a particular object, and is needed for static data initialization. Static constructor has a number of features:

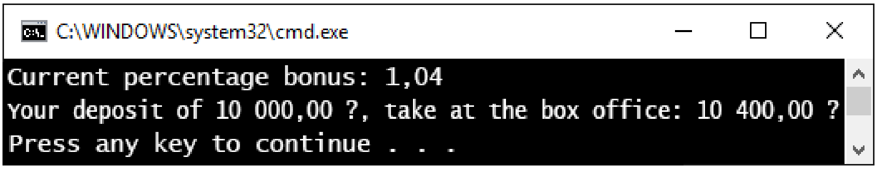
- there can be only one static constructor in the class;
- parameters cannot be passed and access modifiers cannot be specified when declaring a static constructor;
- static constructor is executed only once, regardless of the number of objects created in this class;
- static constructor will be called before any reference to this class (usually before the first call of any member of the class);
- static constructor is never called by any other code, but only by the .NET runtime when loading the class;

- static constructor has access only to the static members of the class.

As an example of the static constructor, let's create a class that describes the bank branches, but this time the static field will contain bonus percentage for deposit arrangement. Each branch will have its own balance.

```
class Bank
{
   private double _currBalance;
   private static double _bonus;

   public Bank(double balance)
   {
      _currBalance = balance;
   }
   static Bank()
   {
      _bonus = 1.04;
   }

   public static void SetBonus(double newRate)
   {
      _bonus = newRate;
   }

   public static double GetBonus()
   {
      return _bonus;
   }

   public double GetPercents(double summa)
   {
      if ((_currBalance - summa) > 0)
      {
            double percent = summa * _bonus;
            _currBalance -= percent;
            return percent;
        }
        return -1;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Bank b1 = new Bank(1000000);
        Console.WriteLine("Current percentage bonus:
                          " + Bank.GetBonus());
        Console.WriteLine("Your deposit of {0:C}, take
                          at the box office: {1: C}",
                          10000, b1.GetPercents(10000));
    }
}
```

Program outcome (Figure 5.4):



C:\WINDOWS\system32\cmd.exe

```
Current percentage bonus: 1,04
Your deposit of 10 000,00 ?, take at the box office: 10 400,00 ?
Press any key to continue . . .
```

Figure 5.4. The use of the static constructor

6. The this keyword

This keyword is a reference to the current instance of the class implicitly present in the class. Let us list some typical uses of this:

1. One of the possible applications of this is the need to resolve the conflict between the names of the method parameters and the names of class fields:

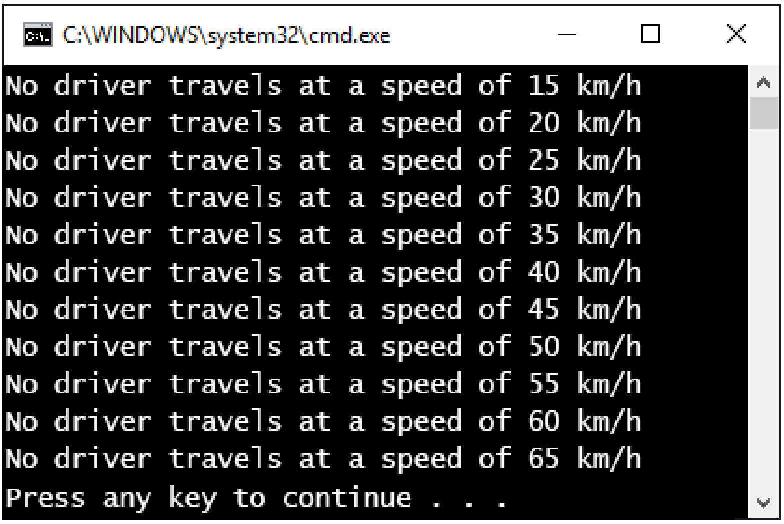
```
class Student
{
   string firstName;
   public Student(string firstName)
   {
      this.firstName = firstName;
   }
}
```

2. Another application of this is an attempt to avoid redundancy when initializing class members. Usually, the main constructor is determined in the class (the main is the constructor with the maximum number of parameters), which contains initialization code. All other constructors call it using this with the necessary parameters in order to avoid code duplication. Let's consider again the example of the Car class, introducing the following changes:

```
class Car
{
   private string _driverName;
```

```
    private int _currSpeed;
    public Car():this("No driver", 0){}
    public Car(string driverName):this(driverName, 0){}
    //Main constructor
    public Car(string driverName, int speed)
    {
        _driverName = driverName;
        _currSpeed = 10;
    }
    public void SetDriver(string driverName)
    {
        _driverName = driverName;
    }
    public void PrintState()//Printing the current data
    {
        Console.WriteLine($"{_driverName} travels
                at a speed of {_currSpeed} km/h.");
    }
    //Speed increase
    public void SpeedUp(int delta)
    {
        _currSpeed += delta;
    }
}

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

Program outcome (Figure 6.1):



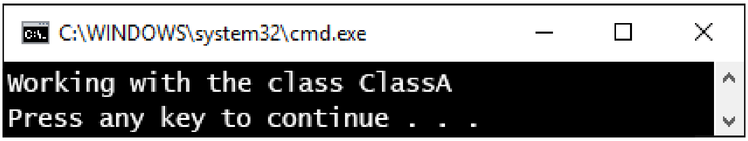
```
No driver travels at a speed of 15 km/h
No driver travels at a speed of 20 km/h
No driver travels at a speed of 25 km/h
No driver travels at a speed of 30 km/h
No driver travels at a speed of 35 km/h
No driver travels at a speed of 40 km/h
No driver travels at a speed of 45 km/h
No driver travels at a speed of 50 km/h
No driver travels at a speed of 55 km/h
No driver travels at a speed of 60 km/h
No driver travels at a speed of 65 km/h
Press any key to continue . . .
```

Figure 6.1. The use of this keyword

3. One more case of using the this keyword consists in passing a reference to the current object to the method as a parameter:

```
public class ClassA
{
    public void MethodA(ClassB obj)
    {
        obj.MethodB(this);
    }
}

public class ClassB
{
    public void MethodB(ClassA obj)
    {
        Console.WriteLine("Working with the class " +
                              obj.GetType().Name);
    }
}

public class Program
{
    public static void Main()
    {
        ClassA a = new ClassA();
        ClassB b = new ClassB();
        a.MethodA(b);
    }
}
```

Code outcome (Figure 6.2):



C:\WINDOWS\system32\cmd.exe

```
Working with the class ClassA
Press any key to continue . . .
```

Figure 6.2. Using the this keyword in methods

4. In the latter case, `this` is used for declaring indexers, the work with which will be discussed in the next lessons.

Using the `this` keyword, you should know that its use is inadmissible in static methods, since they exist only at the class level and are not the parts of the objects of this class.

7. Class methods

Method is a block of code that contains a set of instructions and is located in a class or struct. There are no methods outside the definition of class-functions in C#.

The method definition consists of modifiers, returned value type, followed by the method name, and then the list of arguments (if any) in brackets, followed by the method body, enclosed in curly braces:

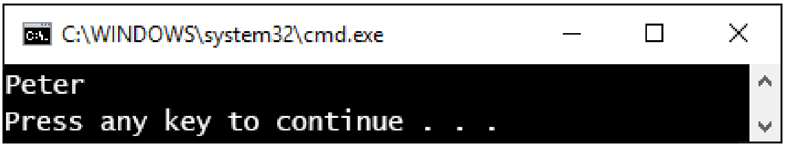
```
[modifiers] return_type method_name ([parameters])
{
   //method body
}
```

Let's add a method to the `Student` class and make all the fields of this class private. Use the practice of class data encapsulation and the provision of public methods for working with this data. Thus you support the principle of data encapsulation and reduce its vulnerability.

```
class Student
{
   private string _firstName = "Peter";

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

class Program
{
```

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

The Code outcome (Figure 7.1):



```
Peter
Press any key to continue . . .
```

Figure 7.1. Example of working with the methods of the class

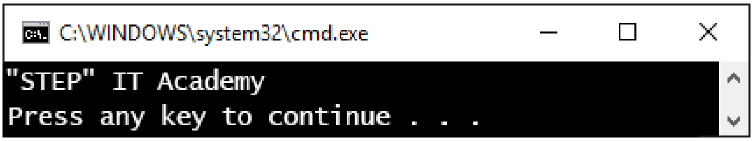
Static methods were introduced especially for working with the static fields. Just as the static fields, these methods belong to the class, not the object. They exclude the possibility of calling through the class object and thus do not work with the non-static fields.

Suppose that all of the students we create will be the students of the STEP IT Academy, and consequently, we add a static field to our `Student` class, which will contain the name of the institution. In order to make the field impossible to be changed, let's make it private and let the `ShowAcademy` static method work with our field in read-only mode.

```
class Student
{
   private static string _academyName="\"STEP\" IT
                                    Academy";

  //previous fields and methods remain unchanged
```

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

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

Program outcome (Figure 7.2):



```
"STEP" IT Academy
Press any key to continue . . .
```

Figure 7.2. Example of working with the static methods

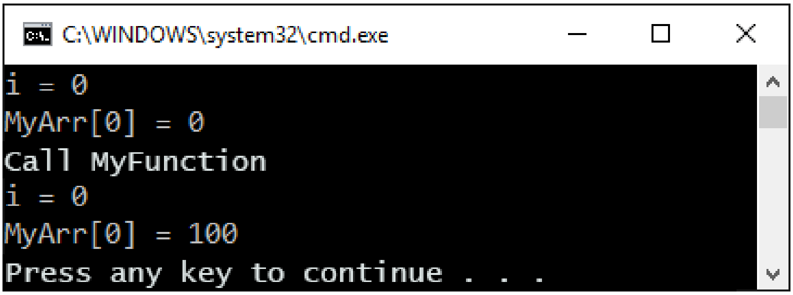
Passing parameters

Arguments can be passed to the method either by value or by reference. When a variable is passed by reference, the called method works with the variable itself, so any changes in the variable within the method will remain valid after its completion. On the other hand, if the variable is passed by value, then the called method creates a copy of this variable, and therefore all the changes in the copy will be lost on the completion of the method. The reference passing is more efficient for complex data types due to the large volume of data that should be copied when passing by value.

In C# all the parameters are passed by value unless otherwise is specified. However, you should be careful with understanding this with respect to reference types. Since the variables of the reference type contains a reference to an object, then the link but not the object itself will be copied when passing a parameter, so the changes made will remain in the object. In contrast, the variables of the value types actually contain data, so the method uses the copies of the data itself that does not cause the change in the original values. Let's consider an example of the above:

```
class Program
{
   static void MyFunction(int[] MyArr, int i)
   {
      MyArr[0] = 100;
      i = 100;
   }

   static void Main(string[] args)
   {
      int i = 0;
      int[] MyArr = { 0, 1, 2, 4 };
      Console.WriteLine("i = " + i);
      Console.WriteLine("MyArr[0] = " + MyArr[0]);
      Console.WriteLine("Call MyFunction");
      MyFunction(MyArr, i);
      Console.WriteLine("i = " + i);
      Console.WriteLine("MyArr[0] = " + MyArr[0]);
   }
}
```

Program outcome (Figure 7.3):



```
C:\WINDOWS\system32\cmd.exe
i = 0
MyArr[0] = 0
Call MyFunction
i = 0
MyArr[0] = 100
Press any key to continue . . .
```

Figure 7.3. Example of working with the method that takes parameters

The return keyword

In addition to the methods that do not return any information, C# also has methods that return data. The syntax of such methods is slightly different: the return value type is added to the method header, and the value itself is returned from the method using the `return` keyword.

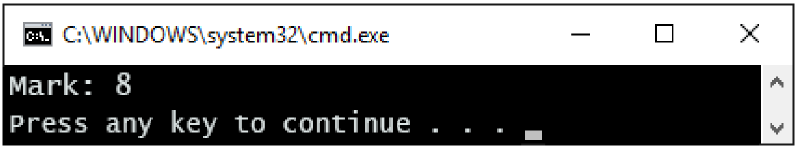
The method can complete its execution in three ways:

1. When control reaches the final brace (this method does not return anything)
2. When control reaches the `return` keyword (without returning value, the return value type of the method is `void`)
3. When control reaches the return keyword (which is followed by the return value, the method returns something)

Let's add a method to our `Student` class that will return his/her mark. The mark itself will be generated randomly (we will use the `Random` class for this purpose).

```
class Student
{
   //previous fields and methods remain unchanged

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

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

Possible program outcome (Figure 7.4):



```
Mark: 8
Press any key to continue . . .
```

Figure 7.4. The use of the return keyword

Method overloading

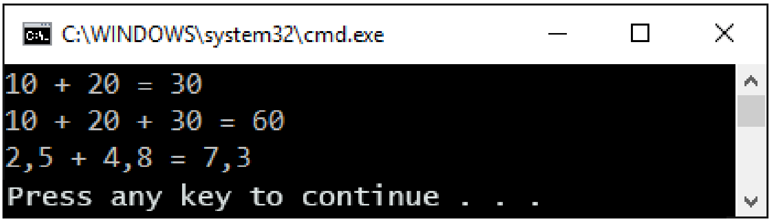
Method overloading is determination of several methods with the same name but with different signatures.

Let's clarify once again that methods should differ only by a signature (number, types, or order of the parameters). The value type returned by the method, as well as the modifiers does not affect overloading!

Let's consider an example, wherein we create a `Mathematic` class having several overloaded methods for adding numbers.

```
class Mathematic
{
   public static int Sum(int a, int b)
   {
      return a + b;
   }
   public static int Sum(int a, int b, int c)
   {
      return a + b + c;
   }
   public static double Sum(double a, double b)
   {
      return a + b;
   }
}

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

Program outcome (Figure 7.5):



```
10 + 20 = 30
10 + 20 + 30 = 60
2,5 + 4,8 = 7,3
Press any key to continue . . .
```

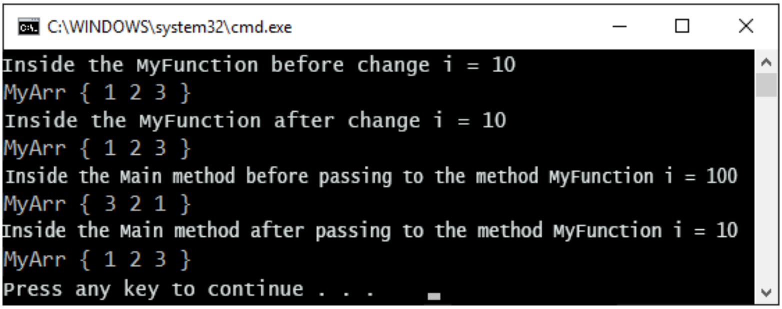
Figure 7.5. Example of method overloading

8. Using ref and out parameters

As mentioned above, the parameters are passed to methods by value (even the reference types, surprisingly). In order to achieve the passing of the parameters through the reference, there are `ref` and `out` keywords. Let's see an example of passing by value.

```
class Program
{
   private static void MyFunction(int i, int[] myArr)
   {
      Console.WriteLine("Inside the MyFunction before
                         change i =" + i);
      Console.Write("MyArr { ");
      foreach (int val in myArr)
         Console.Write(val + " ");
      Console.WriteLine("}");

      i = 100;
      myArr = new int[] { 3, 2, 1 };

      Console.WriteLine("Inside the MyFunction after
                         change i =" + i);
      Console.Write("MyArr { ");
      foreach (int val in myArr)
         Console.Write(val + " ");
      Console.WriteLine("}");
}

static void Main(string[] args)
{
      int i = 10;
      int[] myArr = { 1, 2, 3 };

      Console.WriteLine("Inside the Main method
                         before passing to the method "
                         + "MyFunction i = " + i);
      Console.Write("MyArr { ");
      foreach (int val in myArr)
              Console.Write(val + " ");
      Console.WriteLine("}");

      MyFunction(i, myArr);

      Console.WriteLine("Inside the Main method after
                       passing to the method " +
                       "MyFunction i = " + i);
      Console.Write("MyArr { ");
      foreach (int val in myArr)
        Console.Write(val + " ");
      Console.WriteLine("}");
   }
}
```

As a result of the execution we see that even the array passing is performed by the value, and the original array is not changed in the `Main` method, (Figure 8.1).



```
C:\WINDOWS\system32\cmd.exe
Inside the MyFunction before change i = 10
MyArr { 1 2 3 }
Inside the MyFunction after change i = 10
MyArr { 1 2 3 }
Inside the Main method before passing to the method MyFunction i = 100
MyArr { 3 2 1 }
Inside the Main method after passing to the method MyFunction i = 10
MyArr { 1 2 3 }
Press any key to continue . . .
```

Figure 8.1. Passing the array by value

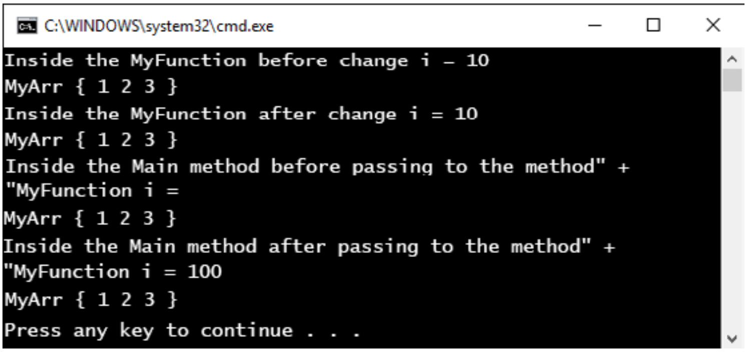
Using the ref modifier

The ref keyword marks the parameters that are passed to the method by reference. Thus, we will manipulate the data declared in the calling method inside the current method. The arguments passed to the method with the ref keyword should be initialized, otherwise the compiler will issue an error message. Let's try to define the parameters of the previous example as a reference and see how the situation will change. Note that even when you call the method, you should specify the ref keyword, informing that the arguments are passed by reference.

```
class Program
{
   private static void MyFunction(ref int i, ref int[] myArr)
   {
      Console.WriteLine("Inside the MyFunction before
                           change i = " + i);
      Console.Write("MyArr { ");
      foreach (int val in myArr)
      Console.Write(val + " ");
      Console.WriteLine("}");

      i = 100;
      myArr = new int[] { 3, 2, 1 };
      Console.WriteLine("Inside the MyFunction after
                           change i = " + i);
      Console.Write("MyArr { ");
      foreach (int val in myArr)
         Console.Write(val + " ");
      Console.WriteLine("}");
   }
```

```
    static void Main(string[] args)
    {
        int i = 10;
        int[] myArr = { 1, 2, 3 };

        Console.WriteLine("Inside the Main method
                          before passing to the
                          method" + "MyFunction i = " + i);
        Console.Write("MyArr { ");
        foreach (int val in myArr)
        Console.Write(val + " ");
        Console.WriteLine("}");

        MyFunction(ref i, ref myArr);
        Console.WriteLine("Inside the Main method after
                          passing to the method" +
                          "MyFunction i = " + i);
        Console.Write("MyArr { ");
        foreach (int val in myArr)
            Console.Write(val + " ");
        Console.WriteLine("}");
    }
}
```

Program outcome has changed (Figure 8.2):



C:\WINDOWS\system32\cmd.exe

```
Inside the MyFunction before change i = 10
MyArr { 1 2 3 }
Inside the MyFunction after change i = 10
MyArr { 1 2 3 }
Inside the Main method before passing to the method" +
"MyFunction i =
MyArr { 1 2 3 }
Inside the Main method after passing to the method" +
"MyFunction i = 100
MyArr { 1 2 3 }
Press any key to continue . . .
```

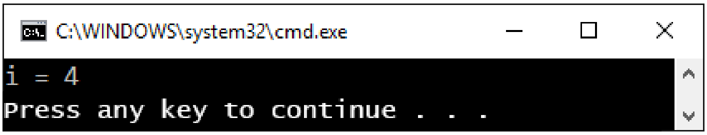
Figure 8.2 The use the ref modifier

Using the out modifier

Parameters marked with the `out` keyword are also used for passing by reference. The difference between `ref` and out is that the parameters are considered output, and the compiler, accordingly, allows not to initialize them before passing to the method, but rather will make sure this parameter is ***necessarily*** initialized in the method (otherwise you will receive an error message — Figure 8.4).

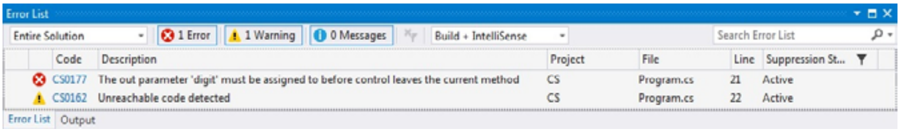
```
class Program
{
   static void Main(string[] args)
   {
      int i;
      GetDigit(out i);
      Console.WriteLine("i = " + i);
   }

   private static void GetDigit(out int digit)
   {
      //return; // Error
      digit = new Random().Next(10);
   }
}
```

Possible program outcome (Figure 8.3):



```
i = 4
Press any key to continue . . .
```

Figure 8.3. The use of the out modifier



Code	Description	Project	File	Line	Suppression St...
CS0177	The out parameter 'digit' must be assigned to before control leaves the current method	CS	Program.cs	21	Active
CS0162	Unreachable code detected	CS	Program.cs	22	Active

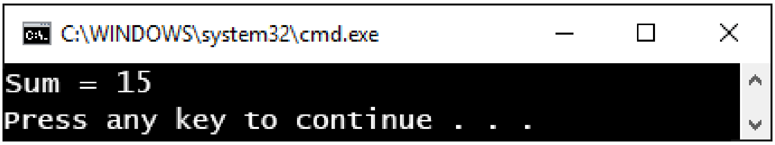
Figure 8.4. Error: no parameter initialization

9. Creating methods with a varying number of arguments

To create a method with a varying number of arguments, an array of value can be passed as a parameter. Let's consider an example of such a method.

```
class Program
{
   static void Main(string[] args)
   {
      Console.WriteLine("Sum = " + Sum(new int[] { 1,
                       2, 3, 4, 5 }));
   }

   private static int Sum(int[] arr)
   {
      int res = 0;
      foreach (int i in arr)
         res += i;
      return res;
   }
}
```

The result of the method is shown in the Figure 9.1:



```
Sum = 15
Press any key to continue . . .
```

Figure 9.1. Method with a variable number of arguments

Although the result is correct, we can see that the approach is crudely vulgar. Therefore, in order to create methods with a variable number of arguments, there is a `params` keyword, which marks the parameter of the method. When using this keyword, you should take into account that the parameter marked with the `params` keyword:

- should be placed last in the list of parameters;
- should refer to a one-dimensional array of any type.

Let's consider the same example, but with the `params` type parameter.

```
class Program
{
   static void Main(string[] args)
   {
     Console.WriteLine("Sum = " + Sum( 1, 2, 3, 4, 5 ));
   }

   private static int Sum(params int[] arr)
   {
     int res = 0;
     foreach (int i in arr)
        res += i;
     return res;

   }
}
```

As a result, we get the same output (Figure 9.1), but the code has become more elegant.

To conclude, it should be noted that call of the method with a varying number of parameters negatively affects performance, because of their permanent placement in the heap of elements and their removal. Therefore, if possible, it is better to define overloaded methods with a different number of arguments without using `params`.

10. Partial types

Partial types are supported only by the compilers of C# and some other languages, but the CLR doesn't know anything about them.

`Partial` keyword tells the C# compiler that the source code of the class can be located in several files. There are two main reasons for the source code splitting into multiple files.

1. ***Version Control*** — class definition may contain a large amount of code. If this class is simultaneously edited by multiple programmers, then they have to somehow combine their results at the end that is very inconvenient.
2. ***Code separators*** — when creating a new Windows Forms or Web Forms project in Microsoft Visual Studio, some files with the source code are created automatically. They are called templates. When using the Visual Studio designer in the process of creating and editing form control elements, Visual Studio automatically generates all the necessary code and places it into separate files. This greatly improves productivity. Before, the automatically generated code was placed in the same file, wherein the programmer wrote his/her source code. The problem was that when accidentally modifying the generated code, the form designer stopped working. Starting with Visual Studio 2005, two source files are created when creating a new Visual Studio project, one of which is designed for the programmer, and the code generated by the forms editor is placed in the other one. Now the probability of accidentally changing the generated code is much smaller.

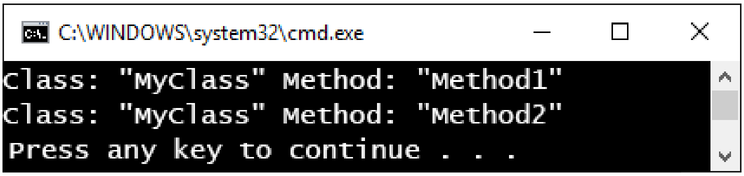
The partial keyword applies to all types of files with class definition. When compiling, the compiler combines these files, and the ready-made class is placed in the resulting assembly file with the .exe or .dll extension. As already mentioned, partial types are implemented only by the C# compiler, so all the files with the source code of the type should be written in one language and compiled into a single module.

For example:

```
//Class1.cs
partial class MyClass
{
   public static void Method1()
   {
      Console.WriteLine("Class: \"MyClass\" Method:
                          \"Method1\"");
   }
}

//Class2.cs
partial class MyClass
{
   public static void Method2()
   {
      Console.WriteLine("Class: \"MyClass\" Method:
                          \"Method2\"");
   }
}

//Program.cs
class Program
{
static void Main(string[] args)
   {
```

```
            MyClass.Method1();
            MyClass.Method2();
        }
    }
```

The result is the same as if a class is declared in a single file (Figure 10.1):



C:\WINDOWS\system32\cmd.exe

```
Class: "MyClass" Method: "Method1"
Class: "MyClass" Method: "Method2"
Press any key to continue . . .
```

Figure 10.1. The use of partial classes

11. Properties

If you have properly studied the C++ language, then you should already ask at this point: "Where are getters and setters?" This section gives us the answer to this question. But first we need to refresh the knowledge of those who have not studied properly, or simply forgotten.

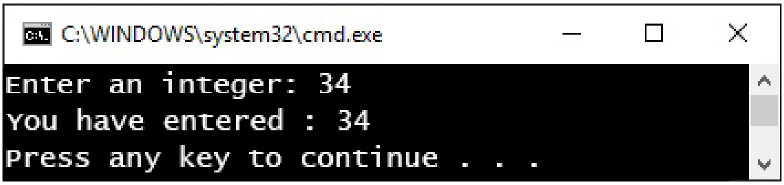
What are the properties?

One of the basic principles of object-oriented programming is encapsulation, and its essence lies in hiding the implementation of a particular class, and providing an interface for working with them. Hiding the implementation is provided by assigning the `private` access modifier to the members of the class. In C++ the interaction interface is provided by certain methods of the class with the `public` access modifier. The methods for getting some values have a common name: getters. The methods for setting some value are called setters.

In C# such an implementation will not lead to errors and will look the following way.

```
using static System.Console;
namespace SimpleProject
{
   class Example
   {
     int _num;
     //method for setting the value of the num variable
     public void Set(int num)
     {
            _num = num;
        }
        //method for getting the value of num variable
        public int Get()
        {
            return _num;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Example example = new Example();
            Write("Enter an integer: ");
            example.Set(int.Parse(ReadLine()));
            Write("You have entered: ");
            WriteLine(example.Get());
        }
    }
}
```

Possible code outcome (Figure 11.1).



```
Enter an integer: 34
You have entered : 34
Press any key to continue . . .
```

Figure 11.1. The use of the methods for working with the class field

In addition, the C# language has a more preferable method of data encapsulation: the use of so-called properties. Property is a simplified representation of the access methods and changes that provides both a possibility to assign values to private fields of the classes regardless of their type, and a possibility of reading their values, implementing any internal logic if necessary. In other words, the properties integrate the capabilities of the two methods that differ in their purpose.

The syntax of properties declaration

The syntax of properties declaration is similar to the syntax of method declaration, but without using the brackets. Usually, property has the `public` access modifier, although, in some cases, it may be `protected`, but cannot be `private`, otherwise, the use of properties would become meaningless. The return type of the property should match the field type, which is encapsulated in this property. The name of the property should match the name of the encapsulated field if agreed, but beginning with a capital letter. The implementation of the property consists of two access methods or the `get` and `set` accessors. The `get` accessor allows getting a value of encapsulated class field, and the `set` provides changing of the value of this field.

```
int _num;
public int Num
{
    get { return _num; }
    set { _num = value; }
}
```

The `get` accessor should include a return statement, which `returns` a class field.

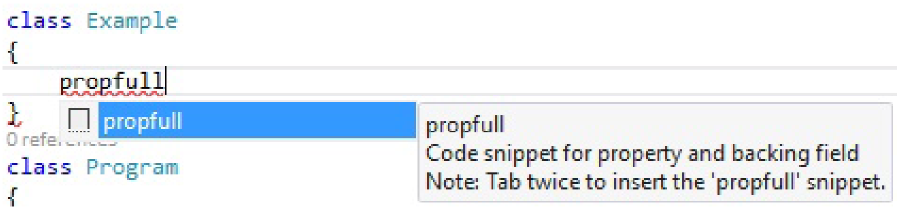
As an assigned value, the `set` accessor uses the `value` token, which is a context-sensitive keyword. In other words, it corresponds to any value assigned to this property. It can be used as a normal identifier outside the property, herewith, name conflicts will not occur.

Let us rewrite the previous example using the properties. The outcome will remain the same (see Figure 11.1).

```
using static System.Console;

namespace SimpleProject
{
   class Example
   {
      int _num;
      public int Num
      {
         get { return _num; }
         set { _num = value; }
      }
   }

   class Program
   {
      static void Main(string[] args)
      {
         Example example = new Example();
         Write("Enter an integer: ");
         example.Num = int.Parse(ReadLine()); // set
         Write("You have entered ");
         WriteLine(example.Num); // get
      }
   }
}
```

As you may notice, it is not specified in the code, which of the accessors should be applied in this situation, because the compiler determines this automatically, based on the way of accessing the property in the code.

There is a simple way to create properties. To do this, when creating a property, you should write a propfull abbreviation (Figure 11.2) and click the Tab key twice.

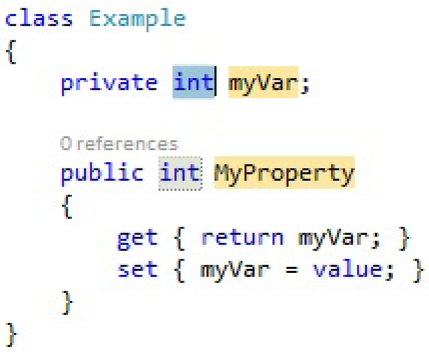


```
class Example
{
    propfull
}
0 references
class Program
{
```

propfull
Code snippet for property and backing field
Note: Tab twice to insert the 'propfull' snippet.

Figure 11.2. Simplified way to create properties (beginning)

After this, it will generate a property template, wherein you can manually specify the required type of the class data field and the names of the field and property if necessary. You can navigate through them by using the Tab key (Figure 11.3).



```
class Example
{
    private int myVar;

    0 references
    public int MyProperty
    {
        get { return myVar; }
        set { myVar = value; }
    }
}
```

Figure 11.3. Simplified way to create properties (continued)

The `get` and `set` accessors are essentially methods. In other words, they can have their own logic. Particularly, the `set` accessor can contain the input values checking code.

For example:

```
set
{
   if(value >= 0 && value <=120)
   {
      _num = value;
   }
}
```

Here, a set value is checked for membership in a range from 0 to 120 inclusively. If everything is fine, then a new value will be assigned to the _num field, otherwise the value will remain the same.

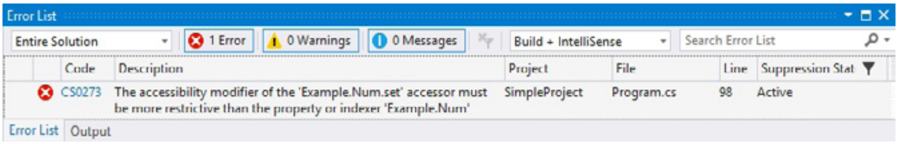
Another point to be noted is an access modifier for each accessor. As you can see, it is not specified explicitly. By default, the access modifiers for both accessors correspond to the level of access of the entire property. The access modifiers for the accessor are specified explicitly only if they impose more restrictions than the modifier of the property itself.

For example:

```
int _num;
   public int Num
   {
      get { return _num; }
      protected set { _num = value; }
   }
```

An error will be generated if you specify an access modifier for each accessor, which would coincide with the property modifier. (Figure 11.4).

```
int _num;
    public int Num
    {
        get { return _num; }
        public set { _num = value; } // Error
    }
```



Error List

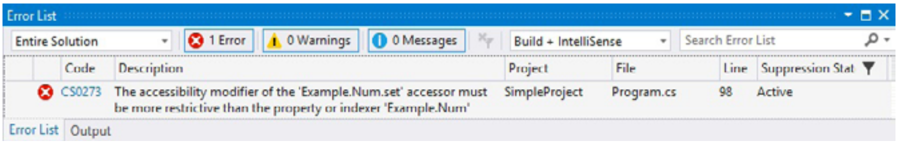
Entire Solution | 1 Error | 0 Warnings | 0 Messages | Build + IntelliSense | Search Error List

	Code	Description	Project	File	Line	Suppression Stat
	CS0273	The accessibility modifier of the 'Example.Num.set' accessor must be more restrictive than the property or indexer 'Example.Num'	SimpleProject	Program.cs	98	Active

Error List | Output

Figure 11.4. Error: access modifier of the accessor should be more restrictive than that of the property

The modifier can be explicitly specified only for one of the accessors, the access modifier of the other accessor will match the modifier of the property. If the access modifier is explicitly specified for the second accessor, that would lead to an error (Figure 11.5).



Error List

Entire Solution | 1 Error | 0 Warnings | 0 Messages | Build + IntelliSense | Search Error List

	Code	Description	Project	File	Line	Suppression Stat
	CS0273	The accessibility modifier of the 'Example.Num.set' accessor must be more restrictive than the property or indexer 'Example.Num'	SimpleProject	Program.cs	98	Active

Error List | Output

Figure 11.5. Error: cannot specify modifiers for both accessors

It is possible to implement only one of the accessor, either `get`, or `set`, and in this case there would be a read-only or write-only property, respectively. For example, a read-only property will look the following way.

```
int _num;
    public int Num
    {
        get { return _num; }
    }
```

In this case, the values of the _num field can be changed only within its class. There are no restrictions on getting the value of this field. A similar result can be achieved by specifying the private access modifier for the set accessor.

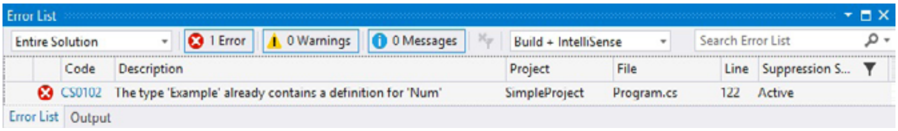
```
int _num;
   public int Num
   {
      get { return _num; }
   }
```

The property can be static. It is necessary in the case when the property is required to set or read the value of the static private field of the class. However, they are called in the same way as the static methods — through the class name. Possible outcome will remain the same (see Figure 11.1).

```
using static System.Console;

namespace SimpleProject
{
   class Example
   {
     static int _num;
     public static int Num
     {
       get { return _num; }
       set { _num = value; }
     }
   }

   class Program
   {
     static void Main(string[] args)
     {
       Write("Enter an integer: ");
```

```
            Example.Num = int.Parse(ReadLine()); // set
            Write("You have entered");
            WriteLine(Example.Num); // get
        }
    }
}
```

It is impossible to split the property into two parts for each accessor, i.e. the following entry is invalid and will generate an error at the compile time (Figure 11.6).



```
Error List
Entire Solution | 1 Error | 0 Warnings | 0 Messages | Build + IntelliSense | Search Error List
  Code     Description                                             Project        File         Line  Suppression S...
  CS0102   The type 'Example' already contains a definition for 'Num'  SimpleProject  Program.cs   122   Active
Error List | Output
```

Figure 11.6. Error: 'Example' type already contains a definition for the Num property

Example use of properties

Before demonstrating an example of using the automatic properties, let's consider another mechanism: syntax of object initialization. This technique greatly simplifies the creation of initialized objects.

The object initializer is specified when declaring a class as initialization of individual properties separated by commas and placed in the braces. Herewith, it is possible to call any constructor of this class before initializing the properties, which is not necessary, but you should keep in mind that the initialization of the properties will override the values of the corresponding fields defined in the constructor. Here is the general form:

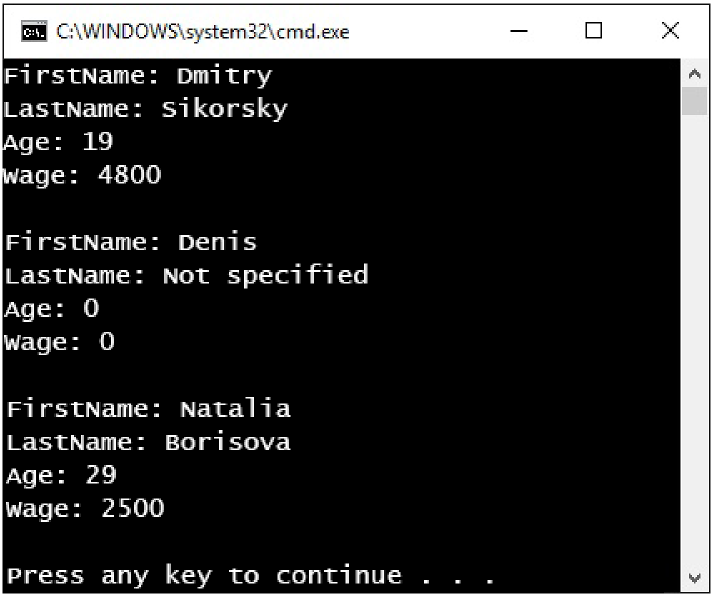
```
new Class_name [(call of any constructor)]
               { Property_name = value,
               Property_name = value}
```

You can initialize a class with public fields in a similar way, but we will reject this possibility because of its unacceptability (think about encapsulation).

In the following example, there is an Employee class with four private fields: first name, last name, age and wage of the employee. Each of the fields of the class has a public property with two accessors. An additional check of the set values is performed in the properties. The first and last names are brought to upper case, the age is verified for membership in the interval of acceptable values, the wage cannot be negative. There are no constructors in this class. The initialization of class fields is performed by the properties due to the syntax of object initialization. Program outcome (Figure 11.7).

```
using static System.Console;
namespace SimpleProject
{
   class Employee
   {
      string _firstName;
      public string FirstName
      {
         get { return _firstName != null ?
                 _firstName : "Not specified"; }
         set { _firstName = value.ToUpper(); }
      }
      string _lastName;
      public string LastName
      {
            get { return _lastName != null ?
                    _lastName : "Not specified"; ; }
            set { _lastName = value.ToUpper(); }
        }

        int _age;
        public int Age
        {
            get { return _age; }
            set { _age = (value > 115 || value < 1)
                    ? 0 : value; }
        }

        float _wage;
        public float Wage
        {
            get { return _wage; }
            set { _wage = value < 0 ? 0 : value; }
        }

        public string Print()
        {
            return $"FirstName: {FirstName}\nLastName:
                   {LastName} \nAge: {Age}\nWage:
                   {Wage}\n";
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Employee emp1 = new Employee { FirstName =
                                "Dmitry", LastName =
                                "Sikorsky", Age = 19,
                                Wage = 4800f };
           Employee emp2 = new Employee();
```

```
emp2.FirstName = "Denis";
         // LastName property was not set
         // Attempt to assign impossible age
         emp2.Age = 120;
         // attempt to set the negative wage
         emp2.Wage = -1000;
         Employee emp3 = new Employee { FirstName =
                         "Natalia", LastName =
                         "Borisova", Age = 29,
                         Wage = 2500f };
            WriteLine(emp1.Print());
            WriteLine(emp2.Print());
            WriteLine(emp3.Print());
        }
    }
}
```


C:\WINDOWS\system32\cmd.exe

```
FirstName: Dmitry
LastName: Sikorsky
Age: 19
Wage: 4800

FirstName: Denis
LastName: Not specified
Age: 0
Wage: 0

FirstName: Natalia
LastName: Borisova
Age: 29
Wage: 2500

Press any key to continue . . .
```

Figure 11.7. Example of using the properties

Auto-properties

Although the properties may contain assigned value checks and change the type of the return value, which was demonstrated in the previous example. However, this doesn't always happen, and the situations are quite often, when the properties are used only for assigning or returning data.

```
int _num;
    public int Num
    {
        get { return _num; }
        set { _num = value; }
    }

    string _firstName;
    public string FirstName
    {
        get { return _firstName; }
        set { _firstName = value; }
    }
```

In such cases, it is recommended to use auto-properties.

What are auto-properties?

When declaring an auto-property, you write the accessors without any implementation, which looks the following way.

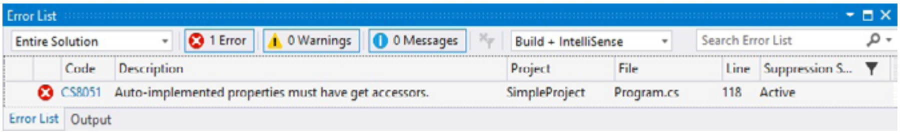
```
public int Num { get; set; }
public string FirstName { get; set; }
```

The feature of auto-property lies in that the compiler automatically creates a hidden field with the `private` access modifier on the basis of this property, and performs the links of this field with the property you wrote. You will further perform manipulations only with the property, and all the work on interaction with a hidden field will be performed by the compiler.

To avoid writing a code of auto-property creation, you can use the "services" of Visual Studio in a way that is already known to you. At this time, you need to write the `prop` abbreviation and press the Tab key twice.

When using auto-properties, it is prohibited to create a property without the `get` accessor. This will lead to an error at the compile time (Figure 11.8).

```
public int Num { get; }
public string FirstName { set; } // Error
```


Error List
Entire Solution | 1 Error | 0 Warnings | 0 Messages | Build + IntelliSense | Search Error List

	Code	Description	Project	File	Line	Suppression S...
✖	CS8051	Auto-implemented properties must have get accessors.	SimpleProject	Program.cs	118	Active

Error List | Output

Figure 11.8. Error: auto-property should have the get accessor

The use of the access modifiers in auto-properties is performed in the same manner as when using the simple properties. Here are some examples of creating the auto-properties for read-only and write-only.

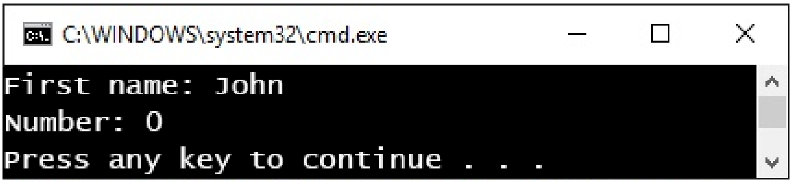
A template of read-only auto-property can also be created by using the Visual Studio, in the same way as described above, but this time it is necessary to write the propg abbreviation and press the Tab key twice.

Initialization of auto-properties

When creating an auto-property, a hidden field created by the compiler is initialized with a default value depending on the data type of the property: `int` — 0, `string` — null, `double` — 0.0, etc. This is shown by the following code.

```
using static System.Console;

namespace SimpleProject
{
    class Example
    {
        public int Num { get; set; }
        public string FirstName { get; set; }
    }
    class Program
    {
       static void Main(string[] args)
       {
           Example example = new Example();

           if (example.FirstName == null)
           {
              example.FirstName = "John";
           }
           WriteLine($"First name: {example.
                 FirstName}\nNumber:{example.Num}");
       }
    }
}
```

Program outcome (Figure 11.9).


C:\WINDOWS\system32\cmd.exe

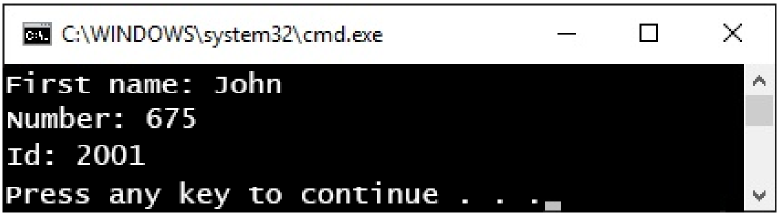
```
First name: John
Number: 0
Press any key to continue . . .
```

Figure 11.9. The default values when working with properties

In C # version 6.0, there is already an ability to initialize the properties by the initial values at declaration. In order to do this, you should write the declaration of these values after declaring the property itself. At that, any expression can be as the initial value. The following code demonstrates this feature — initialization of the identifier depending on the value of the current year.

```
using System;
using static System.Console;
namespace SimpleProject
{

    class Example
    {
        public int Id { get; } = DateTime.Now.Year <
                2000 ? 1001 : 2001;
        public int Num { get; set; } = 675;
        public string FirstName { get; set; } =
                "John";
    }
    class Program
    {
        static void Main(string[] args)
        {
            Example example = new Example();

            WriteLine($"Name: {example.FirstName}\
            nNumber: {example.Num}\nId:
            {example.Id}");
        }
    }
}
```

Program outcome (Figure 11.10).



```
First name: John
Number: 675
Id: 2001
Press any key to continue . . .
```

Figure 11.10. Initialization of the properties by the initial values

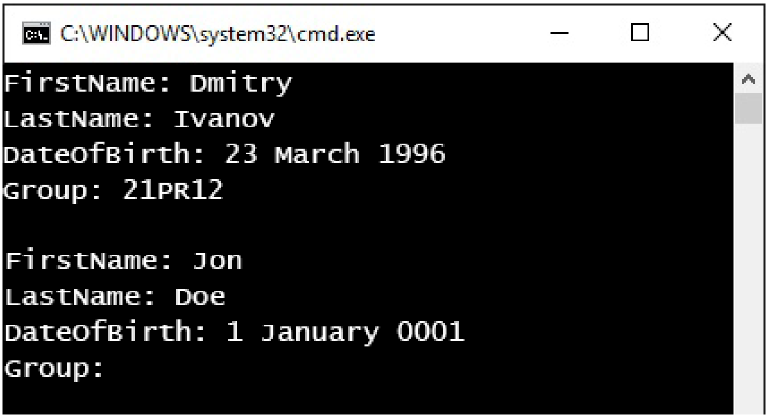
Example use of auto-properties

As an example, let's take the `Student` class and create auto-properties: name, surname, date of birth and the group name. Let's specify default values for the first and last names.

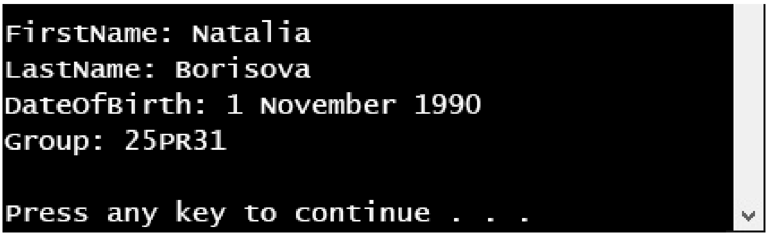
```
using System;
using static System.Console;

namespace SimpleProject
{
   class Student
   {
      public string FirstName { get; set; } = "John";
      public string LastName { get; set; } = "Doe";
      public string Group { get; set; }
      public DateTime DateBirth { get; set; }
      public string Print()
      {
         return $"FirstName: {FirstName}\nLastName:
                {LastName}\nDateOfBirth:
                {DateBirth.ToLongDateString()}\
                nGroup: {Group}\n";
      }
   }
```

```
    class Program
    {
        static void Main(string[] args)
        {
            Student student1 = new Student { FirstName =
                                   “Dmitry”, LastName =
                                   “Ivanov”, DateBirth =
                                   new DateTime(1996,3,23),
                                   Group = “21PR12” };
            Student student2 = new Student();
            Student student3 = new Student { FirstName =
                                   “Natalia”, LastName =
                                   “Borisova”, DateBirth =
                                   new DateTime(1990,11,12),
                                   Group = “25PR31” };
            WriteLine(student1.Print());
            WriteLine(student2.Print());
            WriteLine(student3.Print());
        }
    }
}
```

Program outcome (Figure 11.11).



```
FirstName: Dmitry
LastName: Ivanov
DateofBirth: 23 March 1996
Group: 21PR12

FirstName: Jon
LastName: Doe
DateofBirth: 1 January 0001
Group:
```



```
FirstName: Natalia
LastName: Borisova
DateofBirth: 1 November 1990
Group: 25PR31

Press any key to continue . . .
```

Figure 11.11. The use of auto-properties

Null-conditional operator

In C# version 6.0, there is a new operator for null checking (null-conditional operator), by which the chain of object checking for null values can be written in a more abbreviated form. There are two syntax forms of the null-conditional operator: the first is used in operations with nullable type variables or objects, and looks like a question mark in front of the "point" operator (?.), the second looks like a question mark with the index operator (?[index]), and is used when working with collections of objects accessible by the index.

The conditional statement was used for checking the value for null in the previous versions of C#.

```
Student student = null;
DateTime? date = null;
if (student != null)
{
    date = student.DateBirth;
}
```

If we rewrite this code, but using the operator for null checking, then we would get the same result, but the code will become shorter:

```
Student student = null;
DateTime? date = student?.DateBirth;
```

If you need to check the properties of the class that are also a class for null , then we could get more conventional design, which can be replaced by a quite large ternary operator.

```
Student student = new Student();
string groupName = student == null ? null : student.
Group == null ? null : student.Group.Name;
```

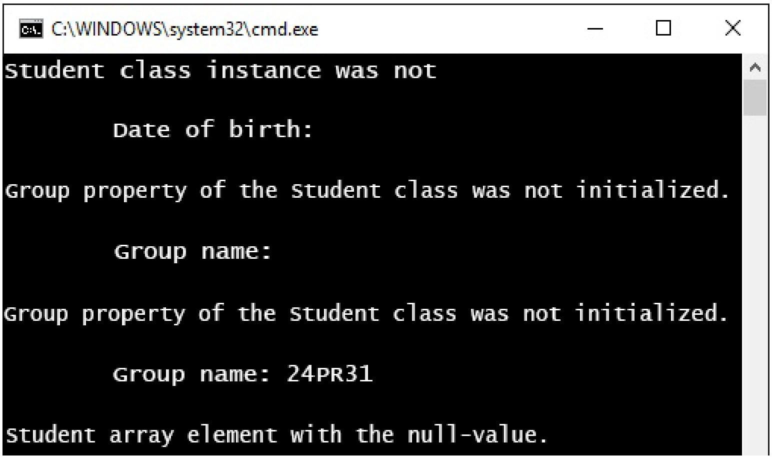
The use of the null-conditional operator provides a more simplified code for sequential checking values for null equality.

```
Student student = new Student();
string groupName = student?.Group?.Name;
```

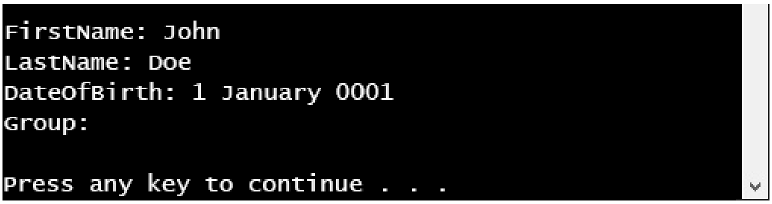
The following code demonstrates all the options of using the null-conditional operator. In the example of working with an array, the operator for checking for null is also supplemented with the ?? operation for creating an instance of the class and initializing the Group property. We will get a result of null, otherwise the call of the Print() method will cause an error at the runtime.

```
using System;
using static System.Console;
namespace SimpleProject
{
    class Group
    {
        public string Name { get; set; }
    }

    class Student
    {
        public string FirstName { get; set; } = "John";
        public string LastName { get; set; } = "Doe";
        public Group Group { get; set; }
        public DateTime DateBirth { get; set; }
        public string Print()
          {
              return $"FirstName: {FirstName}\
                     nLastName: {LastName}\
                     nDateOfBirth: {DateBirth.
                     ToLongDateString()}\nGroup:
                     {Group.Name}\n";
         }
    }

    class Program
    {
        static void Main(string[] args)
        {
            WriteLine("Student class
                       instance was not");
            Student student1 = null;
            DateTime? date = student1?.DateBirth;
            WriteLine($"\n\tDate of birth: {date}");

            WriteLine("\nGroup property of the Student
                       class was not initialized.");

            Student student2 = new Student();
            string groupName = student2?.Group?.Name;
            WriteLine($"\n\tGroup name:
                       {groupName}");
            WriteLine("\nGroup property of the Student
                      class was not initialized.");

            Student student3 = new Student { Group =
                      new Group { Name = "24PR31" } };
            groupName = student3?.Group?.Name;
            WriteLine($"\n\tGroup name:
                      {groupName}");

            WriteLine("\nStudent array element
                      with the null-value.\n");

            Student[] students = { student1,
                      student2, student3 };

            Student student = students?[0] ??
                    new Student { Group = new Group() };

            WriteLine(student.Print());

        }
    }
}
```

Program outcome (Figure 11.12).



```
Student class instance was not

        Date of birth:

Group property of the Student class was not initialized.

        Group name:

Group property of the Student class was not initialized.

        Group name: 24PR31

Student array element with the null-value.
```



```
FirstName: John
LastName: Doe
DateOfBirth: 1 January 0001
Group:

Press any key to continue . . .
```

Figure 11.12. The use of the auto-properties

Many of those who read to the end of this section would think: "How is this section related to the properties?" Of course, this topic should be highlighted earlier, but then you would not have understood the examples given here.

12. Namespace

What is a namespace?

Now let's take a closer look at one of the fundamental concepts of the C# language. No program is managed without this concept. It's all about the so-called namespaces. What is a namespace in C#?

Namespace is a certain area of data declaration that provides a logical relationship of types.

In other words, the namespace is a way of grouping associated types, including classes, structs, delegates, interfaces, enumerations, and other namespaces.

When creating a class in a project file, you can place it in the namespace you will assign to it. If you later decide to include another class in the program (possibly in a different file), that performs some associated task, then you should place it in the same namespace, creating a logical group. Thus, you point to how these classes interact with each other.

If any type (for example, the `Poetry` class) has been declared in its namespace (for example, `Inspiration`), then it indicates that this type will exist peacefully in its namespace, without conflicting with the same types declared in the namespaces other than this.

Frankly speaking, the namespace concept is very lifelike.

For example, my neighbor's name is Alex, and I had a classmate who also has this name. There is also one fellow coworker who (You won't believe!!!) is also named Alex.

The list goes on ...

And while all these people are united by the same name, they cannot be confused, because each of them is limited by the logic of his context. For one of them, it is a class in the school, for another one — the house where I live, and for the third one — the team, in which I work now.

The information about these people can be displayed in the C# syntax in the following way:

- SchoolClass.Alex
- MyNeighborhood.Alex
- TeamAtWork.Alex

Programming is not far behind the real life, and therefore there is a very similar situation here. It is impossible to imagine how many libraries, classes, functions, variables, and other things like this, each one defined with its own identifier, already exist today. No wonder that the programmer working on the creation of the new libraries, classes, and other stuff, can easily give their brainchild a name that had once given to another type. Gigabytes of human memory, unfortunately, are not able to hold such a number of names (I apologize in advance if someone who is reading this tutorial can demonstrate otherwise).

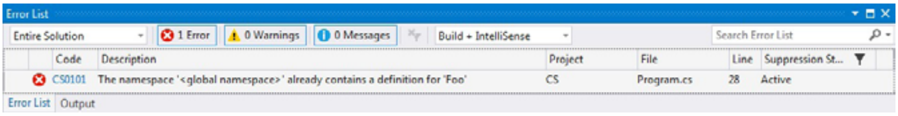
The same situation can be seen in the .NET library. The namespace System.Web.UI.Controls has a Button class, which is also present in the System.Windows.Forms namespace. However, each of them exists within its namespace, not leading to a conflict.

Aims and objectives of the namespace

So, you've probably already guessed that one of the main objectives of creating namespaces is to prevent conflicts of names overlapping. Consider the following example:

```
using System;

class Foo
{
//...
}
class Foo
{
//...
}
```

In response to this code, the compiler issues the following error message (Figure 12.1):



	Code	Description	Project	File	Line	Suppression St...
❌	CS0101	The namespace '<global namespace>' already contains a definition for 'Foo'	CS	Program.cs	28	Active

Figure 12.1. Error: the declaration of classes with the same names

Both classes are declared in a single file, so they are located in the same scope. And since they have the same identifiers, there occurs one of the most common situations — the conflict of names.

To solve this problem, you need to either change the name of one of the classes, or delimit their scopes using namespaces. Of course, in this case, the first method will do like this:

```
using System;

class Foo
{
    //...
}
```

```
class Foo1
{
    //...
}
```

But in another case it may be critical. Then the conflict can be completely eliminated using the namespace:

```
using System;

namespace CSharpNamespace
{
    class Foo
    {
        //...
    }
}
class Foo
{
        //...
}
```

or like this:

```
using System;

namespace CSharpNamespace
{
    class Foo
    {
        //...
    }
}
```

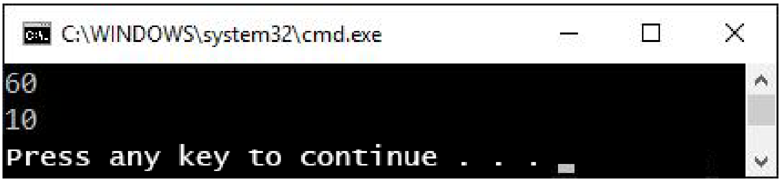
```
namespace CSharpNamespace1
{
    class Foo
    {
        //...
    }
}
```

One more example:

```
using System;

namespace A
{
    class Incrementer
    {
        private int _count;

        public Incrementer(int count)
        {
            _count = count;
        }
        public int MultyIncrement()
        {
            for (int i = 0;   i < 5; i++)
                _count++;
            return _count;
        }
    }

}
namespace B
{
    class Incrementer
    {
        private int _var;
        public Incrementer(int var)
```

```
            {
                _var = var;
            }
            public int AnotherMultyIncrement()
            {
                for (int i = 0; i < 5; i++)
                    _var += 10;
                return _var;
            }
        }

        class Tester
            {
                public static void Main()
                {
                    Incrementer obj1 = new Incrementer(10);
                    Console.WriteLine(obj1.
                                AnotherMultyIncrement());
                    A.Incrementer obj2 =
                                new A.Incrementer(5);
                    Console.WriteLine(obj2.MultyIncrement());
                }
            }
}
```

Program outcome (Figure 12.2):



```
60
10
Press any key to continue . . .
```

Figure 12.2. The use of homonymous classes

In this program, two classes with the same name `Incrementer` are placed into two different namespaces A and

B located in a single file. The entry point of the program is the `Main` method placed in the second of the classes. The instance of the `Incrementer` class from the namespace B is created in the Main method, and the method that increases the value of the field is created for this instance

```
Incrementer obj1 = new Incrementer(10);
Console.WriteLine(obj1.AnotherMultyIncrement());
```

Then another object is created — an instance of the Incrementer class of the namespace A.

```
A.Incrementer ob2 = new A.Incrementer(5);
Console.WriteLine(ob2.MultyIncrement());
```

Pay attention to the entry: a class name is specified in combination with the title of the namespace. Why is it made this way? The space creates a scope for all the objects (types) located within it. The objects of the namespace B are not visible inside the namespace A and vice versa. In order to refer to the type of namespace A in the namespace B, you should explicitly specify the name of the second one.

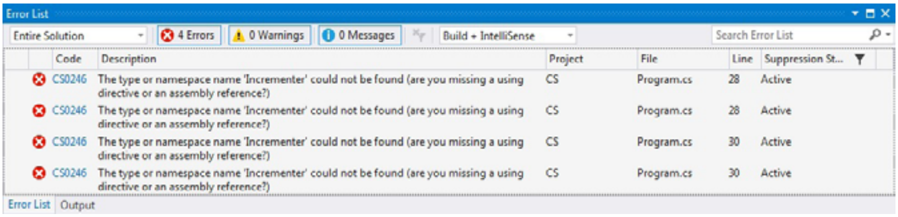
And what will happen if we act differently and would not add the title of the namespace A before the `Incrementer` class name? Let's change the entry and re-run the program.

Now the code looks like this:

```
class Tester
{
    public static void Main()
    {
        Incrementer obj1 = new Incrementer(10);
```

```
            Console.WriteLine(obj1.
                                AnotherMultyIncrement());
            Incrementer obj2 = new Incrementer(5);
            Console.WriteLine(obj2.MultyIncrement());
        }
}
```

And the compiler issues an error message (Figure 12.3):



Error List

Code	Description	Project	File	Line	Suppression St...
CS0246	The type or namespace name 'Incrementer' could not be found (are you missing a using directive or an assembly reference?)	CS	Program.cs	28	Active
CS0246	The type or namespace name 'Incrementer' could not be found (are you missing a using directive or an assembly reference?)	CS	Program.cs	28	Active
CS0246	The type or namespace name 'Incrementer' could not be found (are you missing a using directive or an assembly reference?)	CS	Program.cs	30	Active
CS0246	The type or namespace name 'Incrementer' could not be found (are you missing a using directive or an assembly reference?)	CS	Program.cs	30	Active

Figure 12.3. An error at compile time

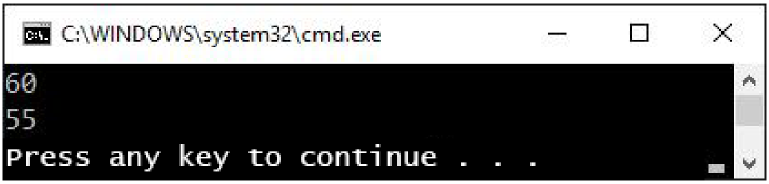
This occurs because the compiler takes the declaration of the second object as an attempt to create another instance of the Incrementer class from the namespace B. And since it has no definition of a method called MultyIncrement, an error is generated.

Suppose the method called MultyIncrement is defined in both namespaces in the relevant Increment classes. Two instances of the Incrementer class followed by calling the MultyIncrement method for both of them are created in the Main method. This is demonstrated in the following code:

```
using System;
namespace A
{
    public class Incrementer
    {
        private int _count;
        public Incrementer(int count)
        {
            _count = count;
        }

        public int MultyIncrement()
        {
            for (int i = 0; i < 5; i++)
                _count++;
            return _count;
        }
    }

}
namespace B
{
   public class Incrementer
    {
        private int _var;

        public Incrementer(int var)
        {
            _var = var;
        }

       public int MultyIncrement()
        {
            for (int i = 0; i < 5; i++)
                _var += 10;
            return _var;
        }
    }

    class Tester
        {
           public static void Main()
           {
```

```
                                Incrementer obj1 = new Incrementer(10);
                                Console.WriteLine(obj1.MultyIncrement());
                                Incrementer obj2 = new Incrementer(5);
                                Console.WriteLine(obj2.MultyIncrement());
                        }
                }
}
```

Program outcome (Figure 12.4):



```
60
55
Press any key to continue . . .
```

Figure 12.4. Calling the homonymous methods in the different classes

This situation will not cause any complaints of the compiler, because in terms of the syntax, it is absolutely correct. However, the question arises whether it is appropriate to create the first namespace and all of its contents, because the `namespace` A is not involved.

The using keyword

In order to include the proper namespace in the program, you should use the `using` keyword followed by the name of the namespace. This line is present in each of your console applications:

```
using System;
```

It is needed in order you can, for example, write text messages to console:

```
public static void Main()
{
      Console.WriteLine("Hello, everybody!");
}
```

The fact is that the Console class is located in the System namespace, and to avoid explicitly specifying the name of the System for each addressing to this class in the program, we connect it via the using directive.

This is how the same code looks without the using directive:

```
public static void Main()
{
      System.Console.WriteLine("Hello, everybody!");
}
```

This is how it goes throughout the entire program. Tiresome, isn't it?

Of course, the using keyword can also be used when creating your own namespaces. For example, our project has two files — Class1.cs and Class2.cs, each one having its namespace, CSharpNamespace1 and CSharpNamespace2, respectively. In the method of the class from the CSharpNamespace2 namespace of the Class2.cs file we want to create an instance of the class from the CSharpNamespace1 namespace of the Class1.cs file. To do this we only need to connect the namespace from another file via using in the Class2.cs file. Here's how it looks:

```
// Class1.cs

using System;
```

```
namespace CSharpNamespace1
{
    class Foo
    {
        //...
    }
}
```

```
// Class2.cs

using System;
using CSharpNamespace1;
namespace CSharpNamespace2
{
    class Class2
    {
        public void Method()
        {
            Foo obj = new Foo();
            }
        }
    }
```

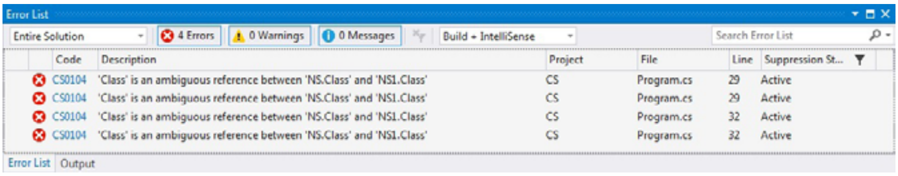
If two namespaces specified in the using operators contain the type of the same name, then it is necessary to use a full (or at least a longer) form of the name, so the compiler would know exactly what type should be used in each case. If you do not, the compiler won't understand to which type of which namespace you are addressing at the moment (Figure 12.5).

```
// Class1.cs
using System;
namespace NS
```

```
{

    public class Class
    {
        public int a = 1;
        public void Print()
        {
            Console.WriteLine("Printing from NS");
        }
    }
}
```

```
// Class2.cs

using System;
namespace NS1
{
   public class Class
    {
        public int b = 2;
        public void Print()
        {
            Console.WriteLine("Printing from NS1");
        }
    }
}
```

```
// Program.cs

using System;
using NS;
using NS1;
namespace M
{
    public class ClassM
    {
        public static void Main()
        {
            Class objA = new Class();
            Console.WriteLine("objA = " + objA.a);
            objA.Print();
            Class objB = new Class();
            Console.WriteLine("objB = " + objB.b);
            objB.Print();
        }
    }
}
```



Error List

Entire Solution | 4 Errors | 0 Warnings | 0 Messages | Build + IntelliSense | Search Error List

Code	Description	Project	File	Line	Suppression St...
CS0104	'Class' is an ambiguous reference between 'NS.Class' and 'NS1.Class'	CS	Program.cs	29	Active
CS0104	'Class' is an ambiguous reference between 'NS.Class' and 'NS1.Class'	CS	Program.cs	29	Active
CS0104	'Class' is an ambiguous reference between 'NS.Class' and 'NS1.Class'	CS	Program.cs	32	Active
CS0104	'Class' is an ambiguous reference between 'NS.Class' and 'NS1.Class'	CS	Program.cs	32	Active

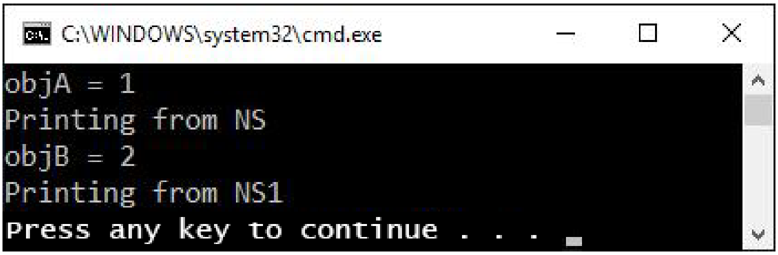
Error List | Output

Figure 12.5. Error: homonymous classes in the different namespaces

To fix these errors, simply add names of the namespaces before the `Class` identifiers:

```
public static void Main()
{
    NS.Class objA = new NS.Class();
    Console.WriteLine("objA = " + objA.a);
    objA.Print();

    NS1.Class objB = new NS1.Class();
    Console.WriteLine("objB = " + objB.b);
    objB.Print();
}
```

Program outcome (Figure 12.6):



```
objA = 1
Printing from NS
objB = 2
Printing from NS1
Press any key to continue . . .
```

Figure 12.6. Explicit namespace declaration in the classes

Namespace declaration

The namespace keyword is used for the namespace declaration. Everything defined within it is enclosed in braces.

```
namespace dataBinding
{
    ...
}
```

The namespace can include classes, structs, delegates, interfaces, enumerations, and other namespaces (you will get acquainted with some of the above C# language tools in the following lessons).

Nested namespaces

Everything is quite simple here: the namespaces can be nested into each other. This creates a hierarchical structure for your types:

```
namespace ITAcademy
{
    namespace ProgrammingDepartment
    {
        namespace CSharp
        {
            namespace Basics
            {
                class MyClass
                {
                    //...
                }
            }
        }
    }
}
```

Each name in the namespace consists of the names of the spaces in which it is embedded. All the names are separated by dots. The first name indicated is the most external one, and then it goes deep into the hierarchy. For example, a full name for the class in the previous example is:

```
ProgrammingDepartment.CSharp.Basics.MyClass
```

A hierarchical arrangement is typical for .NET, and the majority of the platform types have a full name consisting of multiple namespaces.

```
System.Collections.Generic.List<>
System.Windows.Forms.Button
System.IO.StreamWriter
System.Xml.Serialization.XmlSerializationReader
```

Multi-level nested type policy protects from matching the names of your types with the types developed by other companies. If you are developing classes for your company, the Microsoft recommends using at least a double layer of namespaces nesting: the first layer is the name of your company, and the second layer is the name of the technology or software package, which the class belongs to. It looks like this:

```
namespace CompanyName
{
    namespace BankingSerices
    {
        class Client
        {
            //...
        }
    }
}
```

and a full name of the type is:

```
CompanyName.BankingSerices.Client
```

In this example, the external `CompanyName` namespace is the root. It is usually serves only for expanding the scope, and the types are not directly defined in it.

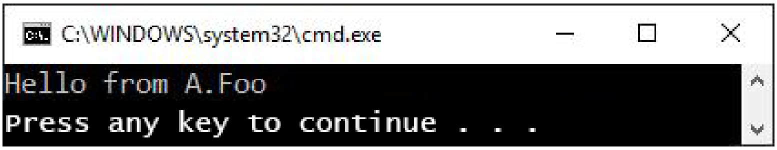
The full namespace name can be used in the definition of the namespace, so the previous example can be rewritten as follows:

```
namespace CompanyName.BankingSerices
{
    class Client
    {
        //...
    }
}
```

As for the namespaces nesting, you need to understand one simple rule: you create the scopes nested into each other, and this means that all types declared in the namespace on a lower level are not available from the enclosing namespaces. In confirmation of this, let's consider an example:

```
using System;
namespace NS
{
    namespace A
    {
        class Foo
        {
            public void Method()
            {
                Console.WriteLine("Hello from A.Foo");
            }

            static void Main()
            {
                Foo obj = new Foo();
                obj.Method();
            }
        }
    }
    class Foo
    {
        public void Method()
        {
            Console.WriteLine("Hello from NS.Foo");
        }
    }
}
```

Program outcome will be shown in the Figure 12.7:



```
Hello from A.Foo
Press any key to continue . . .
```

Figure 12.7. The scope of the namespaces

Decomposition of a namespace into pieces

Imagine a situation: several programmers of your company are working on the development of different classes for the same application. Each of these classes is located in its file, but placed in the same namespace. When these files will be added to the project, they would be compiled into a single assembly.

You can also decompose a namespace into pieces within a single file. Let's consider the following example:

```
// Class1.cs
using System;
namespace A
{
    public class ClassA
    {
        public void Print()
        {
            Console.WriteLine("Printing from A.ClassA");
        }
    }
}
```

```
// Class2.cs
using System;
namespace A
{
    class ClassB
    {
        public void Print()
        {
           Console.WriteLine("Printing from A.ClassB");
        }
    }
}
namespace A
{

    class ClassC
    {

        public void Print()
        {
          Console.WriteLine("Printing from A.ClassC");
        }
    }
}
```

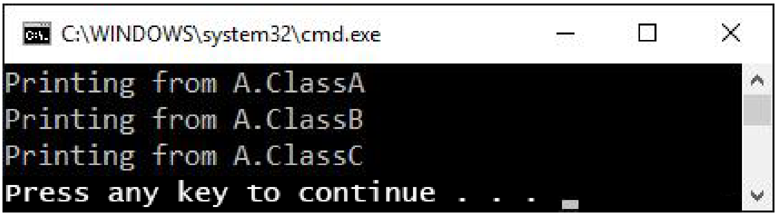
```
// Program.cs

using System;
using A;
namespace B
{
    public class Class
    {
        public static void Main()
        {
            ClassA a = new ClassA();
            a.Print();

            ClassB b = new ClassB();
            b.Print();
```

```
                ClassC c = new ClassC();
                c.Print();
            }
        }
}
```

Here, the namespace A is distributed in two files, but actually is broken into three parts, as it is further decomposed into two parts in the `Class2.cs` file. The operator is included in the `Program.cs` file:

```
    using A;
```

This is necessary to ensure that all the types defined in the namespace A are available in the second namespace B. Here is the program outcome (Figure 12.8):



```
Printing from A.ClassA
Printing from A.ClassB
Printing from A.ClassC
Press any key to continue . . .
```

Figure 12.8. Example of the namespace decomposition into pieces

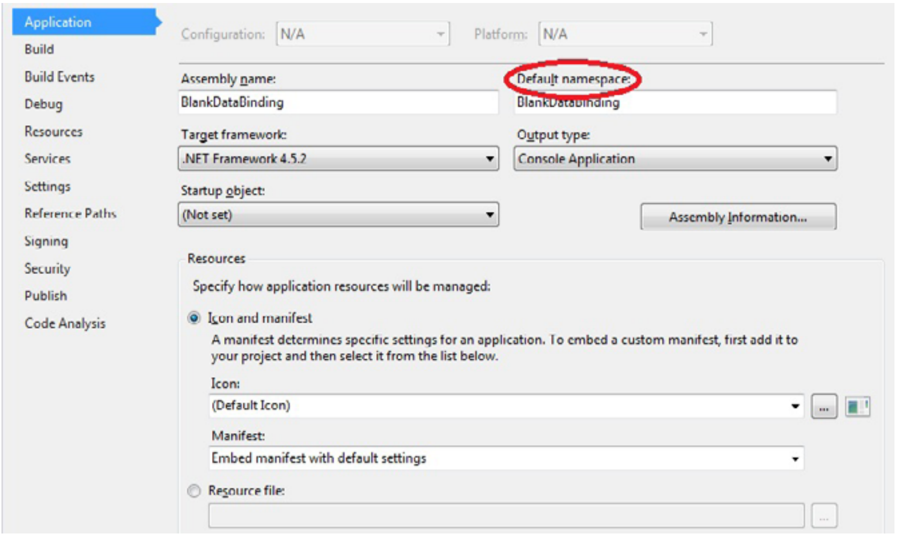
Default namespace

According to the rules of the .NET Framework, all the names should be declared within a namespace.

And what would happen if we do not assign any namespace for our types?

In this case, the default namespace will be assigned to them, which will have the same name as the project name.

If you are not happy with it, then you can change its name to the one desired. All you need is to use the Default namespace option and specify the name you want in the Application tab of the project properties (Figure 12.9).



Configuration: N/A
Platform: N/A

Assembly name: BlankDataBinding
Default namespace: BlankDataBinding
Target framework: .NET Framework 4.5.2
Output type: Console Application
Startup object: (Not set)

Assembly Information...

Resources
Specify how application resources will be managed:
Icon and manifest
A manifest determines specific settings for an application. To embed a custom manifest, first add it to your project and then select it from the list below.
Icon: (Default Icon)
Manifest: Embed manifest with default settings
Resource file:

Figure 12.9. Setting the default namespace

If you do so, then each subsequent file you will add to the project will be automatically placed in the namespace with this name.

Using directive

The `using` keyword can be used not only for connecting the required namespace, but also for creating a namespace alias.

Why is it convenient? As we mentioned earlier, since the namespace can have any nesting depth, then the size of the identifier can be quite bulky, for example:

```
using System.Windows.Forms.DataVisualization.Charting;
```

To avoid reentering this name every time, we actually create one more, briefer name for the namespace in the following way:

```
using alias = full name of the namespace;
```

```
using WFCharting = System.Windows.Forms.
                   DataVisualization.Charting;
```

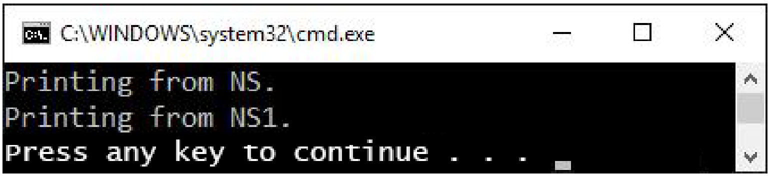
In the future, it is desirable to address the type with the "::" qualifier. In case the type with the name that coincides with the alias is defined in this namespace, there will be no conflict. Let's consider the following example:

```
// Class1.cs
using System;
namespace NS
{
    public class Class
    {
        public void Print()
        {
            Console.WriteLine("Printing from NS.");
        }
    }
}
```

```
// Class2.cs
using System;

namespace NS1
{
    public class Class
    {
        public void Print()
        {
            Console.WriteLine("Printing from NS1.");
        }
    }
}
```

```
// Program.cs

using System;
using X = NS;
using Y = NS1;
namespace M
{

    public class ClassM
    {
        public int m = 3;

        public static void Main()
        {
            //X.Class objA = new X.Class(); Error
            X::Class objA = new X::Class();
            objA.Print();
            Y::Class objB = new Y::Class();
            objB.Print();
        }
    }

    public class X
    {
         //...
    }
}
```

Program outcome (Figure 12.10):



C:\WINDOWS\system32\cmd.exe

```
Printing from NS.
Printing from NS1.
Press any key to continue . . .
```

Figure 12.10. The use of the qualifier

This example shows a project consisting of several files: `Class1.cs`, `Class2.cs`, `Program.cs`.

In each file, the code is enclosed in its namespace. Aliases for the namespaces from the other two files are assigned in the `Program.cs` file:

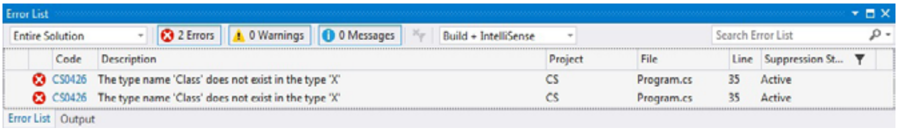
```
using X = NS;
using Y = NS1;
```

And this is where the types from the namespaces `NS` and `NS1` are addressed via the aliases. The logic goes that the situation is common, but ...

The `X` class is declared in the namespace `M` of the `Program.cs` file, and its name matches the alias assigned to the namespace `NS`:

```
public class X
{
    //...
}
```

If we addressed the `X` identifier in the body of the `Main` method, the compiler would have regarded it as a reference to the class with the same name. It is impossible to address the `Class` from the namespace X via the point; an error is shown in the Figure 12.11.


Error List

Entire Solution | 2 Errors | 0 Warnings | 0 Messages | Build + IntelliSense | Search Error List

	Code	Description	Project	File	Line	Suppression St...
⊗	CS0426	The type name 'Class' does not exist in the type 'X'	CS	Program.cs	35	Active
⊗	CS0426	The type name 'Class' does not exist in the type 'X'	CS	Program.cs	35	Active

Error List | Output

Figure 12.11. Error at the compile time

The situation is saved by the "::" qualifier.

```
X::Class objA = new X::Class();
```

Applying using to connect static members

When accessing the static members of the class, you should specify the class name, for example:

```
Console.WriteLine("Hello");
```

Such actions had to be repeated throughout the program, but in the C # version 6.0 it became possible to import static classes via the using keyword, it looks in the following way:

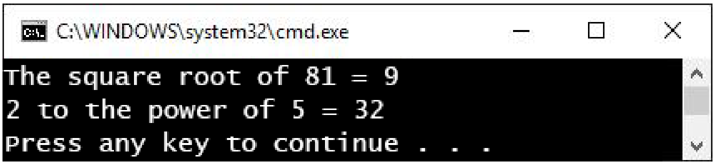
```
using static System.Console;
```

After that, the call of methods of static classes can be carried out without specifying the class name, as shown below:

```
using static System.Console;
using static System.Math;
class Program
{
    static void Main()
    {
        WriteLine("The square root of 81 = " +
                  Sqrt(81));
```

```
            WriteLine(("2 to the power of 5 = " + Pow(2, 5));
        }
    }
```

Code outcome:



```
The square root of 81 = 9
2 to the power of 5 = 32
Press any key to continue . . .
```

Figure 12.12. Applying the using static directive

The `using static` directive imports only available static members and nested types that are declared in this class. Inherited members are not imported.

The `using` keyword is also used to create the `using` statement, which helps to ensure the proper treatment of class objects that implement the `IDisposable` interface, but we will study them in one of the following lessons.

13. Home task

1. Describe the `Article` struct containing the following fields: product code; product name; price.
2. Describe the `Client` struct containing the following fields: client code; full name; address; telephone; number of orders by the client; total amount of orders.
3. Describe the `RequestItem` struct containing the following fields: commodity; number of commodity units.
4. Describe the `Request` struct containing the following fields: order code; client; order date; a list of ordered products; order price (implement using the calculated property).
5. Describe the `ArticleType` enumeration that defines the types of commodities, and add the appropriate field to the Article struct from task 1.
6. Describe the `ClientType` enumeration that defines customer's importance, and add the appropriate field to the `Client` struct from the task №2.
7. Describe the `PayType` enumeration that defines the form of order payment by the client, and add an appropriate field to the `Request` struct from the task №4.
8. Create a class that describes a student. Provide it with the following: last name, first name, patronymic, group, age, array (jagged) of marks on programming, administration and design. Also add methods for working with the enumerated data: the ability to set/get a mark, get an average mark for a specified subject, the printout of student data.
9. Develop an application "7 Wonders of the World", wherein every wonder will be represented as a separate class.

Create an additional class that contains an entry point. Distribute an application into project files and provide classes interaction using a namespace.

10. Develop an application, which compares population of three capitals of different countries. Moreover, a country should be defined as a namespace, and a city — as a class in this space.