# Object-Oriented Programming

Before we dive into this module about OOP, we strongly recommend going through our [module 1 about C# basics](#s). You will find many valuable pieces of information, which will help you significantly to follow along with this module.

# Classes and Constructors

The class is the root of the word classification. When we create our class we systematically arrange information and behavior into a meaningful entity. We don’t use classification only in the software development, we are doing that in a real life as well. So, this only explains how classification is important in a software development process. The classes are reference data types, and if you want to learn more about data types you can visit [our article in module 1 about C# basics](#d).

## Adding New Elements in a Solution Explorer

Even though we can create our new classes in the Program.cs file, it is the much better solution to create a new class file. To do that, we need to right-click on our project name, choose Add and then New Item:



Then, we need to choose a class file and add it a name:



## Defining Classes and How to Use Them

In C#, to define a class we need to use the class keyword. All the data and methods occur in the class body between two curly braces:

public class Student

{

private string \_name;

private string \_lastName;

public string GetFullName()

{

return \_name + ' ' + \_lastName;

}

}

The classes body contains two private fields (variables in a class are called fields) name and lastName (if you are not familiar with the access modifiers keywords: private, public etc. you can read more [about it in our module 1 about C# basics](#r)), and one public method GetFullName.

As we know from our module 1 C# basics, the class is a reference type, so to initialize it we need to use the new keyword:

class Program

{

static void Main(string[] args)

{

Student student = new Student();

}

}

Now with the student object we can access the data from the Student class.

It is very important not to confuse the terms class and object. The class is a type definition but an object is an instance of that type. We can have several object instances of the same class.

## Constructors

When we use the new keyword to create an object, the CLR uses the class definition to construct that object for us by calling a constructor method.

A constructor is a special method that has the same name as a class, doesn’t return any value (not even void) and can take parameters. It runs automatically when we create an instance of a class. So, every time we use the new keyword to instantiate a class, we are calling a constructor of that class.

Every class must have a constructor. If we don’t write one, the compiler automatically generates one for us. That type of constructor is called a **default constructor**. A default constructor will reset all the data inside a class, to their default values. So, in our example, the fields name and lastName will have an empty string as a value at a beginning.

We can write our own default constructor as well:

public class Student

{

private string \_name;

private string \_lastName;

public Student()

{

\_name = string.Empty;

\_lastName = string.Empty;

}

public string GetFullName()

{

return \_name + ' ' + \_lastName;

}

}

## Constructor Overloading

Our classes are not restricted on having just one constructor method. We can create more of them in a single class:

public class Student

{

private string \_name;

private string \_lastName;

public Student()

{

\_name = string.Empty;

\_lastName = string.Empty;

}

public Student(string name, string lastName)

{

\_name = name;

\_lastName = lastName;

}

public string GetFullName()

{

return \_name + ' ' + \_lastName;

}

}

Now we have two options to instantiate our class, first one with the default values (which we don’t have to write) and the overloaded one, which provides us with a possibility to set the values of our fields:

class Program

{

static void Main(string[] args)

{

Student student = new Student(); //default constructor

Student student1 = new Student("John", "Doe");//overloaded constructor

Console.WriteLine(student1.GetFullName());

}

}

One important thing to have in mind. If we create our own constructor for a class, the compiler won’t create a default one for us. So if we want to have a default one and the overloaded one, we must create both of them.

## Partial Classes

In a real-world project, our class can be pretty large due to its own high functionality. That kind of classes could become less readable and tough to maintain. To avoid that, we can use partial classes.

A partial class is nothing more than a part of a single class. To define partial classes, we need to use the partial keyword in each file:

partial class Student

{

private string \_name;

private string \_lastName;

public Student()

{

\_name = string.Empty;

\_lastName = string.Empty;

}

}

partial class Student

{

public Student(string name, string lastName)

{

\_name = name;

\_lastName = lastName;

}

public string GetFullName()

{

return \_name + ' ' + \_lastName;

}

}

# Properties

A property is a member that provides a flexible tool to read and write the value of a private field. We use them as a public data members but actually, they are specific methods called accessors.

In this article, we are going to talk more about properties and how to use them in C#.

## Property Syntax

The syntax of a property declaration can be represented in the following way:

Access\_Modifier Type PropertyName

{

get

{

//read actions

}

set

{

//write action

}

}

As we can see, a property can contain two blocks of code. The get block contains statements that execute when we read from a property. The set block contains statements that execute when we write to a property:

public class Student

{

private string \_name;

private string \_lastName;

public string Name

{

get { return \_name; }

set { \_name = value; }

}

public string LastName

{

get { return \_lastName; }

set { \_lastName = value; }

}

public Student(string name, string lastName)

{

\_name = name;

\_lastName = lastName;

}

public string GetFullName()

{

return \_name + ' ' + \_lastName;

}

}

In the example above we see that our private fields are now exposed through the properties. If we want to read the value of the \_name field all we have to do is to call a Name property with the student object. The same applies to the \_lastName field. Moreover, if we want to set a value to our fields, all we have to do is to call a set block of our properties:

class Program

{

static void Main(string[] args)

{

Student student = new Student("John", "Doe");

string name = student.Name; //call to a get block of the Name property

string lastName = student.LastName; // call to a get block of the LastName property

student.Name = "David"; //call to a set block of the Name property

student.LastName = "Dauni"; // call to a set block of the LastName property

}

}

Our properties can have a complex code inside get or set blocks. They are not limited only to read a value or just to write a value. We can use conditions or method calls etc. in the get or set blocks:

public int X

{

get

{

return \_x;

}

set

{

\_x = CheckValue(value);

}

}

private int CheckValue(int val)

{

//code execution in here

}

## Read-Only and Write-Only Properties

We can declare a property that only has a get block and not the set. That kind of property is called Read-Only property. If we create a read-only property, we can only read the value of a private field. If we try to set it, the compiler will throw an error:

public string Name

{

get { return \_name; }

}



In the same way, as we can create a read-only property, we can create a write-only property. That type of property has only the set block and not the get. Of course, we can only set the values with this type of property and not to read it:

public string Name

{

set { \_name = value; }

}



## Property Accessibility

We can specify an access modifier for our property (public, private…) if we want to restrict its availability. But in C# we can even override the accessibility of get or set accessors. So, what we can do is to declare a public property which has the public get accessor and private set accessor. If our property is a public one, we don’t have to add the public keyword for the get accessor, it is going to be public anyway:

public string Name

{

get { return \_name; }

private set { \_name = value; }

}



This means that we can read in all the classes from our Name property, but we can set it only inside the Student class.

When we use an accessor overriding inside the property, we must pay attention to the following rules:

* We can change the accessibility level of only one accessor. There is no point in having both accessors modified. If we want to modify both accessors, we should just modify the property access level.
* We can’t use access modifier on the get or set blocks that are less restrictive of the access modifier applied on a property itself. So, if our property is private, there is no point in having the get or set blocks public.

## Auto-Implemented Properties

If no additional logic is required in a property accessor, we can use the auto-implemented properties for more readable and concise way of declaring properties. The auto-implemented property consists only of the get and set keywords, nothing more:

public string Name { get; set; }

public string LastName { get; set; }

When we declare the properties like this, the compiler creates a private field for us, which could be accessed only through the property’s get or set accessors.

So in our example instead of:

private string \_name;

public string Name

{

get { return \_name; }

set { \_name = value; }

}

We can just write:

public string Name { get; set; }

In the VisualStudio tool we are even going to get a suggestion to use an auto property:

