# Object-Oriented Programming

Before we dive into this module, 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 word “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 the same in real-life situations. So as it is important in a real-life, it is important in a software development. The classes are reference data types, and if you want to learn more about data types you can visit [our module about C# basics](#d).

## Adding New Elements in Solution Explorer

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



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. The class consists of members. All the class members are defined in the class body between two curly braces:

public class Student

{

private string \_name;

private string \_lastName;

public string GetFullName()

{

return \_name + ' ' + \_lastName;

}

}

We see that the body contains two private fields (variables in the class body are called fields) \_name and \_lastName and one public method GetFullName (if you are not familiar with the access modifiers: private, public etc. you can read more about them in our [module 1 about C# basics](#dd)).

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 the student object can access the members from the Student class. For now, we have only one method inside the Student class and we can call it with student1.GetFullName() syntax. This will return an empty string, but this will be fixed as soon as we introduce constructors.

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:

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

Student student10 = new Student();

Student student20 = new Student();

Student student30 = new Student();

## 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.

The constructor is a special method that has the same name as the class it is defined in, 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. This type of constructor is called a **default constructor**. A default constructor will set all the data inside a class, to their default values (assigned values if we don’t assign them). 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 gives us the ability 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());

}

}

There is 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 with so many lines of code. That kind of classes could become less readable and tough to maintain. To avoid that, we can use partial classes. Partial classes have even more advantages because multiple developers can work on the same class at the same time. Furthermore, we can create a partial method inside those classes as well.

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 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 used 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. It is quite common to create a read-only property inside our class. What we want with it is to set it with the constructor method and then to use its value throughout the entire class, but never to set its value **outside the constructor**. 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. It is not a common case to create write-only properties. Of course, if we need it, we can only set the values with this type of property and not 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 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 public get or set block.

## 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 Visual Studio, we are even going to get a suggestion to use an auto property:



# Static Methods, Static Classes, and Extension Methods

In this article, we are going to talk about static members in C#, when and why to use them.

## About Static Methods

When we define a method in a class, it belongs to that class, and every instance of that class will be able to access it. One class can have many such methods. But there are some methods that are independent of the specific class instance. That kind of methods is called “static methods”. So, the static methods are the methods which don’t belong to an instance of a class, can interact only with other static elements and have the static keyword in the method description.

Let’s take the Sqrt() method for example. This method calculates the square root of a number, and we don’t have to instantiate the Math class (which the Sqrt belongs in) because this method is a static method:

int number = 4;

Console.WriteLine(Math.Sqrt(number));

So, why is the Sqrt method a static method and not a nonstatic one?

Well, the Sqrt accepts only one argument and it is enough to do its job. We provide an argument number and the method returns a square root of that number. We didn’t mention the Math class at all. That’s because we don’t have to. The Math class doesn’t provide any support to the Sqrt method to do its job. It only provides a space for the Sqrt method to reside in.

When we have a case like this one, it is usually a good solution to create a method as a static one.

## Working with a Static Method

To call a static method, as we said, we don’t need an instance of a class. We can call it with the following syntax: ClassName.MethodName(arguments…);

So, when we want to use the Sqrt method or any other method from the Math class, we can call it like this: Math.Sqrt(16);

## Creating a Field by Using the Const Keyword

If we prefix our field with the const keyword, we can declare such a field where its value can never change. The keyword const is short for constant. A constant member is defined at compile time and it can’t be modified at runtime.

We can create a const variable in the following way: AccessModifier const Type Name = Value ;



## Static Class

In C#, next to static methods we can declare static classes as well. The static class can contain only the static members. Its purpose is to act as a holder for the utility methods and fields. There is no point in instantiating this type of classes by using the new keyword. Furthermore, we can’t do that at all. But we can create a default constructor as long as it is a static one. Any other type of constructor is illegal:

public static class TestClass

{

private static int number;

static TestClass()

{

number = 54;

}

}

## About Extension Methods and How to Use Them

Let’s suppose that we want to add a new feature to the string type, for example, the FirstLetterUpperCase functionality that always makes the first letter of a string with upper case. We can write a normal method for that purpose:

public static string FirstLetterUpperCase(string word)

{

char letter = Char.ToUpper(word[0]);

string remaining = word.Substring(1);

return letter + remaining;

}

static void Main(string[] args)

{

string word = "football";

string newWord = FirstLetterUpperCase(word);

}

But, as we can see, we need to send a word as a parameter every time and to accept a value every time as well. This is not a wrong approach but we can do it even better. There's where the extension methods come in.

An extension method enables us to extend an existing type with additional static methods. We must create that kind of methods inside a static class and they have the first parameter prefixed with the “this” keyword.

But why do we have to place a prefix in front of the first parameter?

Because that parameter is an indicator that tells to the compiler which type we extend.

So here is the previous example but with the extension method:

public static class StringExtensions

{

public static string FirstLetterUpperCase(this string word)

{

char letter = Char.ToUpper(word[0]);

string remaining = word.Substring(1);

return letter + remaining;

}

}

class Program

{

static void Main(string[] args)

{

string word = "football"

.FirstLetterUpperCase();

Console.WriteLine(word);

Console.ReadKey();

}

}

Excellent.

We are done with the static members and now we have a great tool in our toolbox that we can use while developing our C# applications.

# Anonymous Types and Nullable Types

In this article, we are going to talk about anonymous classes, how to create them, and why they are useful. Moreover, we are going to talk about nullable types and how to use them with the value types and what properties we have with the nullable types.

## Anonymous Classes

An anonymous class is a class that does not have a name. This sound strange but sometimes an anonymous class can be useful, especially when using query expressions.

Let’s see what we mean by that.

We can create an anonymous class simply by using the new keyword in front of curly braces:

myAnonymousObj = new { Name = "John", Age = 32 };

This class contains two properties the Name and the Age. The compiler will implicitly assign the types to the properties based on the types of their values. So what this means basically is that the Name property will be of the type string and the Age property of the type int.

But now, we can ask, what the type of the myAnonymousObj is? And the answer is that we don’t know, but this is the point of anonymous classes. But in C# this is not a problem, we can declare our object as an implicitly typed variable by using the var keyword:

var myAnonymousObj = new { Name = "nesto", Age = 32 };

The var keyword causes the compiler to create a variable of the same type as the expression that we use to initialize that object. So let’s see a couple of examples with well-known types:

var number = 15; // the number is of type int

var word = "example"; //the word is of type string

var money = 987.32; //the money is of type double

We can access to the properties of our anonymous object the same way we did with regular objects:

Console.WriteLine($"The name of myAnonymousObject is {myAnonymousObj.Name}, the age is {myAnonymousObj.Age}");

## Nullable Types

The null value is useful for initializing reference types. So, it is logical that we can’t assign the null value to the value type because the null is itself a reference. The following statement will throw an error:



However, C# provides us with a modifier that we can use to declare a value type as a nullable value type. We can use the ? to indicate that value type is nullable:

int? number = null;

We can still assign an integer value to our nullable value type:

int? number = null;

int another = 200;

number = 345;

number = another;

This is all valid. But if we try to assign a value of our nullable type to the variable of an int type, we are going to have a problem:

int? number = null;

int another = 200;

another = number; //this is the problem

This makes sense if we consider that the variable number might contain the null but the variable another can’t contain the null at all.

## Properties of Nullable Types

The nullable types expose a few properties which can come in handy while working on our projects. The HasValue property indicates whether a nullable type contains a value or it is a null. The Value property enables us to retrieve the value of the nullable type if it is not a null:

int? number = null;

number = 234; //comment this line to print out another result

if(number.HasValue)

{

Console.WriteLine(number.Value);

}

else

{

Console.WriteLine("number is null");

}

# Structures

In the previous articles, we have learned about classes, how to use them and how to create an object as an instance of a class. In this article, we are going to talk about structures which are similar to classes but have some differences as well.

## Working with Structures

A structure is a value type, in the opposite of a class which is a reference type, and it has its own fields, methods, and constructors like a class.

Maybe you didn’t realize, but we have worked with structures in our previous articles, especially in [module 1 C# basics](#rr). Int, double, decimal, bool type etc. are all aliases for the structures System.Int32, System.Int64 etc. In the following table, we can see the primitive types and what are they built from (class or structure):

|  |  |  |
| --- | --- | --- |
| Keyword | Type | Created from |
| bool | System.Boolean | Structure |
| byte | System.Byte | Structure |
| decimal | System.Decimal | Structure |
| double | System.Double | Structure |
| float | System.Single | Structure |
| int | System.Int32 | Structure |
| long | System.Int64 | Structure |
| object | System.Object | **Class** |
| sbyte | System.SByte | Structure |
| short | System.Int16 | Structure |
| String | System.String | **Class** |
| uint | System.UInt32 | Structure |
| ulong | System.UInt64 | Structure |
| ushort | System.UInt16 | Structure |

## Structure Declaration

To declare our own structure, we need to use the struct keyword followed by the name of the type and then the body of the structure between two curly braces:

public struct Time

{

private int \_hours, \_minutes, \_seconds;

}

We can create our own constructor to initialize our private fields:

public struct Time

{

private int \_hours, \_minutes, \_seconds;

public Time(int hours, int minutes, int seconds)

{

\_hours = hours;

\_minutes = minutes;

\_seconds = seconds;

}

public void PrintTime()

{

Console.WriteLine($"Hours: {\_hours}, Minutes: {\_minutes}, Seconds: {\_seconds}");

}

}

To access our structure we can use this syntax:

static void Main(string[] args)

{

Time time = new Time(3, 30, 25);

time.PrintTime();

Console.ReadKey();

}

## Differences Between Classes and Structures

* The structure is a value type, while the class is a reference type
* We **can’t** declare our own default constructor in a structure. That’s because a structure is always generating a default constructor for us. In a class, we **can** create a default constructor because a class won’t generate then one for us
* We can initialize fields in our structure by creating a non-default constructor, but we must initialize all of the fields inside that constructor. It is not allowed to left a single field without a value:



With a class, this is not a case

* In a class, we can initialize instance fields at their point of declaration. In a structure, we cannot do that:



* An instance of a class lives on a heap memory while the instance of a structure lives on a stack
* In a structure, we can create a non-default constructor, but nevertheless, the compiler will always generate the default one. This is not the case with a class.

## When to Use Structure Instead of a Class

The general rule that we can follow is that our structures need to be small and simple types and above all immutable. For anything else, we should use a class.

Why is immutability so important?

Well let’s take a look at this example:

class Test

{

public int Number { get; set; }

public Test(int number)

{

Number = number;

}

}

class Program

{

static void Main(string[] args)

{

Test test = new Test(10);

Console.WriteLine(test.Number);

ChangeNumber(test);

Console.WriteLine(test.Number);

}

public static void ChangeNumber(Test test)

{

test.Number = 45;

}

}

If we inspect the result, we will see printed out 10 and 45. And that is the correct result. But if we change our Test class to be a structure and then inspect the result, we will see 10 and 10.

This can lead to the confusion and problems as well, because the consumer may expect that the ChangeNumber method would modify the Number property because we allowed it in the code. But if we create properties or fields immutably (as read-only in a structure) then we can avoid this kind of confusion. The consumer can assign values to the properties by calling the constructor method but after that those properties need to stay immutable.

# Enumeration

Besides the structures, C# supports another value type Enumeration. In this article, we are going to talk more about that value type.

## Working with Enumerations

Suppose we need to represent days in a week in our C# project. We can use an integer number to represent every single day in a week (from 0 to 6), and even though that would work just fine it is not readable at all. This is where enumerations excel.

To declare enumeration we can use the following syntax:

public enum DaysInWeek

{

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday,

Sunday

}

After we have declared our enumeration, we can use it in exactly the same way as any other type:

static void Main(string[] args)

{

DaysInWeek monday = DaysInWeek.Monday;

Console.WriteLine(monday); // It is going to print out Monday

Console.ReadKey();

}

As we can see, we must write DaysInWeek.Monday and not just Monday because all enumeration literal names are in scope of their enumeration type.

## Choosing Enumeration Literal Values

Internally, an enumeration type assigns the integer value to every element inside that enumeration. Those numbers start at 0 and increase by 1 for every other element. In our previous example, we print out the value that matches with the exact element of an enumeration. But we can print the integer value as well by casting it into its underlying type:

static void Main(string[] args)

{

DaysInWeek monday = DaysInWeek.Monday;

Console.WriteLine((int)monday); //prints out the 0

Console.ReadKey();

}

If we prefer, we can assign a specific integer constant to the enumeration elements:

public enum DaysInWeek

{

Monday=1,

Tuesday,

Wednesday,

Thursday, Friday,

Saturday,

Sunday

}

If we do it like this, the Monday will have the value 1 and all the others will be increased by one (Tuesday=2, Wednesday=3…). But we can assign a random value to each of the elements:

public enum DaysInWeek

{

Monday=10,

Tuesday=20,

Wednesday=35,

Thursday=48,

Friday=74,

Saturday=12,

Sunday=154

}

Of course it is always a better way to assign integer values with the equal progression (1, 2, 3… or 10, 20, 30…).

## Choosing an Enumerations Underlying Type

When we declare an enumeration, the compiler assigns integer values to all of the elements. But we can change that. We can provide a different type right after the name of an enumeration:

public enum DaysInWeek: short

{

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday,

Sunday

}

By doing this, we save our memory because the int type is taking more memory than the short, and we don’t need for our example, greater capacity of the short data type.

# Inheritance

Inheritance is one of the three key concepts in an object-oriented programming. We can use inheritance to avoid repetition when different classes have a number of features in common and are related to each other.

In this post, we are going to talk about inheritance, why is it important and what we can use it for.

## Using Inheritance

We can define inheritance between two classes by using the following syntax:

class DerivedClass: BaseClass

{

...

}

The derived class inherits from the base class, thus all the non-private members of the base class become the members of the derived class as well. In C#, a derived class can inherit only from one base class. But we can chain an inheritance from one class to another:

class DerivedClass: BaseClass

{

}

class DerivedSubClass: DerivedClass

{

}

What this means is that DerivedSubClass inherits from the DerivedClass and from the BaseClass as well, because DerivedClass inherits from the BaseClass. That way, we can share the class features between multiple classes, even though the one class can inherit only from one base class.

So, let’s create some basic inheritance structure:

public class Writer

{

public void Write()

{

Console.WriteLine("Writing to a file");

}

}

public class XMLWriter: Writer

{

public void FormatXMLFile()

{

Console.WriteLine("Formating XML file");

}

}

public class JSONWriter: Writer

{

public void FormatJSONFile()

{

Console.WriteLine("Formating JSON file");

}

}

In this example, the XMLWriter and JSONWriter classes have they own methods but both of them share the Write() method from the base class Writer.

So, if we create an object of type XMLWriter, we will be able to access its own method and the method from the base class:

class Program

{

static void Main(string[] args)

{

XMLWriter xmlWriter = new XMLWriter();

xmlWriter.FormatXMLFile();

xmlWriter.Write();

}

}

It goes the same for the JSONWriter class.

## Calling Constructors from the Base Class

From the derived classes, we can access the constructor of a base class. This is used quite common, due to initialization of some properties that are shared between derived classes. We can use the base keyword to execute this:

public class Writer

{

public string FileName { get; set; }

public Writer(string fileName)

{

FileName = fileName;

}

public void Write()

{

Console.WriteLine("Writing to a file");

}

}

public class XMLWriter: Writer

{

public XMLWriter(string fileName)

:base(fileName)

{

}

public void FormatXMLFile()

{

Console.WriteLine("Formating XML file");

}

}

public class JSONWriter: Writer

{

public JSONWriter(string fileName)

:base(fileName)

{

}

public void FormatJSONFile()

{

Console.WriteLine("Formating JSON file");

}

}

class Program

{

static void Main(string[] args)

{

XMLWriter xmlWriter = new XMLWriter("xmlFileName");

xmlWriter.FormatXMLFile();

xmlWriter.Write();

Console.WriteLine(xmlWriter.FileName);

JSONWriter jsonWriter = new JSONWriter("jsonFileName");

jsonWriter.FormatJSONFile();

jsonWriter.Write();

Console.WriteLine(jsonWriter.FileName);

}

}

## Accessing Classes

The inheritance hierarchy means that our XMLWriter (or JSONWriter) class is a special type of the Writer, it has all the Writer’s non-private members, and additional features declared inside the XML(JSON)Writer class. But there are some limitations to this hierarchy. Let’s see the following example:

XMLWriter xml = new XMLWriter("file.xml");

Writer writer = xml;

writer.Write(); //ok Write is part of the Writer class

writer.FormatXML(); //error FormatXML is not part of the Writer class

This means if we refer to the XMLWriter or JSONWriter object with the Writer object, we can just access the methods declared inside the Writer class.

There is one more limitation. We can’t assign a higher rank object to a lower rank object:

Writer writer = new Writer("any name");

XMLWriter xml = writer; //error

But we can solve this problem by using the “as” keyword:

XMLWriter xml = new XMLWriter("any name");

Writer writer = xml; //writer points to xml

XMLWriter newWriter = writer as XMLWriter; //this is ok now because writer was xml

newWriter.FormatXMLFile();

## Declaring Methods with the New Keyword

In the real world project, we often need to have so many different functionalities, and that usually leads to the existence of many different methods, properties etc. Sometimes it is pretty hard to come up with the unique and meaningful name for our identifiers, especially if we have the inheritance hierarchy. Sooner or later we are going to try to reuse a name that is already in use by one of the classes in the higher hierarchy level. If it comes to that (we have two methods with the same name in derived and base class) we are going to receive a warning:



## Using the New Keyword

A method in a derived class hides a method in a base class with the same signature. So, as you see in the picture above, our method SetName exists in the XMLWriter class and Writer class. Since the XMLWriter class inherits from the Writer class it hides an implementation of the SetName method from the Writer class.

Although our code will compile and run, we should take this warning seriously. It can happen that another class inherits from the XMLWriter class and implements the SetName method. The developer may expect to execute the SetName method from the Writer class (because XMLWriter inherits from the Writer) but this is not a case. The SetName method from the Writer class is hidden by the SetName method from the XMLWriter class.

If we find ourselves in this kind of situation the best way is to change the method signatures. But if we are sure that we want a behavior like this, we can use the new keyword. The new keyword will simply tell the compiler that we are hundred percent sure in what we are doing and that we don’t want a warning message to appear anymore:

public class Writer

{

public string FileName { get; set; }

public Writer(string fileName)

{

FileName = fileName;

}

public void Write()

{

Console.WriteLine("Writing to a file");

}

public void SetName()

{

Console.WriteLine("Setting name in the base Writer class");

}

}

public class XMLWriter: Writer

{

public XMLWriter(string fileName)

:base(fileName)

{

}

public void FormatXMLFile()

{

Console.WriteLine("Formating XML file");

}

public new void SetName()

{

Console.WriteLine("Setting name in the XMLWriter class");

}

}

Now we don’t have a warning message any more.

## Declaring Methods with the Virtual Keyword

Sometimes, we don’t want to hide an implementation of a method from a base class with the same signature as a method from a derived class. What we want is to provide an opportunity for different implementation of a method with the same signature in a derived class. So, we want to override our method from a base class with the method inside a derived class.

A method that is intended to be overridden is called a virtual method. When we talk about overriding and hiding, we need to be clear with those terms. The hide means that we want completely to hide the implementation of a method from the base class, but the override means that we want a different implementation of a method from a base class.

To create a virtual method we use the virtual keyword:

public class Writer

{

public string FileName { get; set; }

public Writer(string fileName)

{

FileName = fileName;

}

public void Write()

{

Console.WriteLine("Writing to a file");

}

public void SetName()

{

Console.WriteLine("Setting name in the base Writer class");

}

public virtual void CalculateFileSize()

{

Console.WriteLine("Calculating file size in a Writer class");

}

}

## Declaring Methods with the Override Keyword

If we declare a method as a virtual in our base class, we can create a method in a derived class with the keyword override to declare another implementation of that method:

public class XMLWriter: Writer

{

public XMLWriter(string fileName)

:base(fileName)

{

}

public void FormatXMLFile()

{

Console.WriteLine("Formating XML file");

}

public new void SetName()

{

Console.WriteLine("Setting name in the XMLWriter class");

}

public override void CalculateFileSize()

{

Console.WriteLine("Calculating file size in the XMLWriter class");

}

}

If we want, we can call an original implementation of that method in a derived class by using the base keyword:

public class XMLWriter: Writer

{

...

public override void CalculateFileSize()

{

base.CalculateFileSize();

Console.WriteLine("Calculating file size in the XMLWriter class");

}

}

All this inheritance actions, and different method implementations with the mentioned keywords has its own unique name **polymorphism.**

## Rules to Follow While Working With Polymorphic Methods

There are some important rules which we need to follow when declaring polymorphic methods by using the virtual and override keywords:

* We can’t declare a virtual method as private. Its purpose is to be exposed to a derived class, so making it private is meaningless. Similarly, overridden methods can’t be private because a derived class can’t change the protection level of a method that it inherits
* The signatures of virtual and overridden methods must be identical
* We can override only a virtual method. If we try to override a method that has no virtual keyword, we will get an error
* If we don’t use the override keyword we are not overriding the method we are just hiding it. If this is the behavior we want, we should use the new keyword
* An overridden method is a virtual one as well, so it can be overridden in a further derived class

# Interfaces

Inheriting from a class is a powerful mechanism, but the real inheritance power comes from an interface. An interface provides the members that a class that inherits from the interface must implement.

We can look at the interface as a contract which states that a class that implements an interface must implement all the members from it.

## Defining an Interface

To define an interface we need to use the interface keyword. It is quite similar to defining a class just we use another keyword. Inside that interface, we specify our members without access modifier and implementation. So, we just provide a declaration for members, an implementation is a job for the class that implements that interface:

interface InterfaceName

{

returnType methodName(paramType paramName...);

}

## Implementing an Interface

To implement an interface, we declare a class or structure that inherits from the interface and implements **all the members** from it:

class ClassName: InterfaceName

{

//member implementation

}

Let’s see all of this through the example:

public interface IWriter

{

void WriteFile();

}

public class XmlWritter: IWriter

{

public void WriteFile()

{

Console.WriteLine("Writing file in the XmlWriter class.");

}

}

public class JsonWriter: IWriter

{

public void FormatFile()

{

Console.WriteLine("Writing file in the JsonWritter class.");

}

}

As we can see, after our classes inherit from an interface, they must implement the member WriteFile(). Otherwise, we would get a compiler error.

When we implement an interface, we must ensure to provide method implementation by following this rules:

* The method names and return types must match exactly
* Any parameters must match exactly
* All the methods must be public during implementation. This is only not the case with the explicit interface implementation(we will talk about that a little later)

A class can inherit from a class and implement an interface at the same time. But if this is a case, we must specify a base class first and then an interface comma separated:

public class FileBase

{

public virtual void SetName()

{

Console.WriteLine("Setting name in the base Writer class.");

}

}

public class XmlWritter: FileBase, IWriter

{

public void WriteFile()

{

Console.WriteLine("Writing file in the XmlWriter class.");

}

public override void SetName()

{

Console.WriteLine("Setting name in the XmlWriter class.");

}

}

public class JsonWriter: FileBase, IWriter

{

public void WriteFile()

{

Console.WriteLine("Writing file in the JsonWritter class.");

}

public override void SetName()

{

Console.WriteLine("Setting name in the JsonWriter class.");

}

}

## Referencing Classes Through Interfaces

In the same way that we can reference an object by using a class variable, we can define an object by using an interface variable:

XmlWriter writer = new XmlWriter();

writer.SetName(); //overridden method from a base class

writer.WriteFile(); //method from an interface

As we can see, all the methods are available through the writer object. But let’s now use an interface object for referencing action:

IWriter writer = new XmlWriter();

writer.WriteFile(); //method from an interface

writer.SetName(); //error the SetName method is not part of the IWriter interface

If we use an interface to create an object, we can access only those members declared in that interface.

As we mentioned above, the interface provides a contract for the class that inherits from it. And this is a great advantage of using interfaces, we can always be sure when a class inherits from our interface it will implement all of its members.

But the interface implementation has even more advantages. One of them is object decoupling.

## Using an Interface to Decouple Classes

When one class depends on another class those classes are coupled. This is something we want to avoid because if something changes in class A and Class B depends heavily on Class A, there is a great possibility that we would have to change a Class B as well. Or at least, we won’t be sure if Class B still works properly. Consequently, we want from our classes to be loosely coupled or “decoupled”.

Let’s see what would happen if we create our classes as strongly coupled:

public class XmlFileWriter

{

private XmlWriter \_xmlWriter;

public XmlFileWriter(XmlWriter xmlWriter)

{

\_xmlWriter = xmlWriter;

}

public void Write()

{

\_xmlWriter.WriteFile();

}

}

This XmlFileWriter is a class which has a purpose of writing to an xml file. Now we can instantiate our XmlWriter class, send the object through the XmlFileWriter constructor and call the Write method:

class Program

{

static void Main(string[] args)

{

XmlWriter xmlWriter = new XmlWriter();

XmlFileWriter fileWriter = new XmlFileWriter(xmlWriter);

fileWriter.Write();

}

}

Ok, everything works great for now.

But we have a couple of problems in here. Our XmlFileWriter class is strongly coupled to the XmlWriter class. If we change the WriteFile method inside the XmlWriter class, we must change it in the XmlFileWriter class as well. So, the change in one class leads to change in another. That’s not how we want our code to work.

Another thing. We surely want to have the same behavior for our JsonWriter class. We can’t use this XmlFileWriter (because it accepts only the XmlWriter object), we must create another class and repeat all of our actions. This is pretty bad as well.

Finally, we can ask ourselves, if we really need two classes for the same job. Why can’t we use just one? Well, that’s where interfaces come in.

Let’s modify the XmlFileWriter class:

public class FileWriter

{

private readonly IWriter \_writer;

public FileWriter(IWriter writer)

{

\_writer = writer;

}

public void Write()

{

\_writer.WriteFile();

}

}

Excellent. This is so much better.

Now our class name tells us that this class doesn’t write only xml files. Furthermore, we are not restricting our constructor to accept just XmlWiter class, but all the classes that inherit from the IWriter interface. Our method WriteFile can’t be renamed now because of our interface IWritter, which states that all classes must implement a method with an identical name. We can see now that FileWriter class are decoupled from the XmlWriter or from the JsonWriter, and that we can send objects of both classes to the FileWriter class:

class Program

{

static void Main(string[] args)

{

XmlWriter xmlWriter = new XmlWriter();

JsonWriter jsonWriter = new JsonWriter();

FileWriter fileWriter = new FileWriter(xmlWriter);

fileWriter.Write();

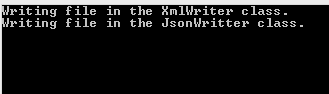
fileWriter = new FileWriter(jsonWriter);

fileWriter.Write();

Console.ReadKey();

}

}



Isn’t this so much better?

Now we have one class that does its job for any class that inherits from the IWriter interface.

This feature is well known as a Dependency Injection.

## Working with Multiple Interfaces

A class can inherit just from one base class, but it can implement multiple interfaces. The class must implement all the methods defined in those interfaces:

public interface IFormatter

{

void FormatFile();

}

public class XmlWriter: FileBase, IWriter, IFormatter

{

public void WriteFile()

{

Console.WriteLine("Writing file in the XmlWriter class.");

}

public override void SetName()

{

Console.WriteLine("Setting name in the XmlWriter class.");

}

public void FormatFile()

{

Console.WriteLine("Formatting file in XmlWriter class.");

}

}

## Explicitly Implementing an Interface

As we already said, a class can implement more than one interface. It’s not unusual that two of those interfaces have a method with the same name, but we still need to implement them in our class. To do that we do not implement a method as we did before, but we need to state the name of the interface first and then the name of a method with parameters:

public interface Interface1

{

void MethodExample();

}

public interface Interface2

{

void MethodExample();

}

public class ExampleClass: Interface1, Interface2

{

void Interface1.MethodExample()

{

Console.WriteLine("");

}

void Interface2.MethodExample()

{

Console.WriteLine("");

}

}

As we can see, we are not using an access modifier in the method implementation.

# Abstract Classes

Different classes may implement the same interface, and that is the common case in programming. What is common as well is that the method from that interface can have the same implementation in those classes. That could be a signal that we are doing something wrong. We don’t want to repeat the code in our classes, but to reuse the common implementation.

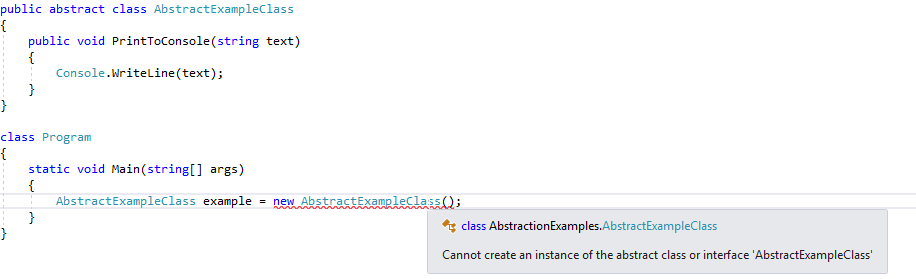
To fix this, we can extract this common implementation to a base class, and make our classes implement a base class and then make the base class implement an interface. This will solve our problem, but it is not a complete solution.

Why is that?

The problem is that now we can create an instance of our base class, which holds nothing except the common implementation of a method (or methods). This doesn’t make any sense. A class that contains only the common implementation should have a sole purpose to be inherited from.

## Creating Abstract Classes

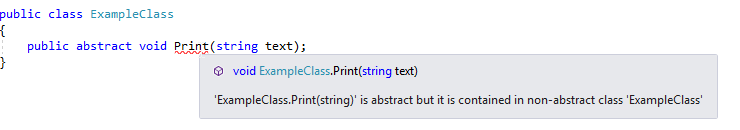
To create an abstract class, we use the abstract keyword. The only purpose of the abstract class is to be inherited from and it cannot be instantiated:



An abstract class can contain abstract methods. An abstract method doesn’t contain implementation just a definition with the abstract keyword:

public abstract void Print(string text);

As we could see from a previous picture, an abstract class doesn’t have to have any abstract member but the more important thing is if a class have at least one abstract member, that class must be an abstract class. Otherwise, the compiler will report an error:



## Sealed Classes

If we want to prevent our class to be inherited, we need to use the sealed keyword. If anyone tries to use a sealed class as a base class, the compiler will throw an error.

# Generics

C# provides generics to help us remove the need for casting, to improve type safety and make it easier to create generic classes and generic methods. To create a generic class, we need to provide a type between angle brackets:

public class CollectionInitializer<T>

{

...

}

The T in this example acts as a placeholder for a type we want to work with. We need to provide that type once we instantiate this generic class. So let’s see this with a simple example:

public class CollectionInitializer<T>

{

private T[] collection;

public CollectionInitializer(int collectionLength)

{

collection = new T[collectionLength];

}

public void AddElementsToCollection(params T[]elements)

{

for(int i=0; i<elements.Length; i++)

{

collection[i] = elements[i];

}

}

public T[] RetrieveAllElements()

{

return collection;

}

public T RetreiveElementOnIndex(int index)

{

return collection[index];

}

}

And to use this generic class:

class Program

{

static void Main(string[] args)

{

CollectionInitializer<int> initializer = new CollectionInitializer<int>(5);

initializer.AddElementsToCollection(5, 8, 12, 74, 13);

int[] collection = initializer.RetrieveAllElements();

int number = initializer.RetreiveElementOnIndex(3);

foreach (int element in collection)

{

Console.WriteLine(element);

}

Console.WriteLine();

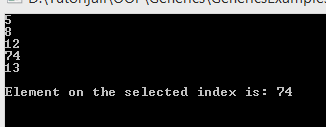
Console.WriteLine($"Element on the selected index is: {number}");

Console.ReadKey();

}

}

As we can see in our CollectionInitializer class, we need to provide the type which we want to work with. Then, we can just call the methods implemented within our generic class. Of course, we didn’t implement safety checks (if we send more elements than the array length is etc) for a sake of simplicity. Now we can see the result:



The type parameter T can be any legal C# identifier, although the lone character T is commonly used.

Of course, we can send any type of data to our generic class now:

class Program

{

static void Main(string[] args)

{

CollectionInitializer<Student> initializer = new CollectionInitializer<Student>(2);

initializer.AddElementsToCollection(new Student { Name="John", Age=25 }, new Student { Name="Jane", Age=24 });

Student[] collection = initializer.RetrieveAllElements();

Student student = initializer.RetreiveElementOnIndex(1);

foreach (Student element in collection)

{

Console.WriteLine(element.Name + " " + element.Age);

}

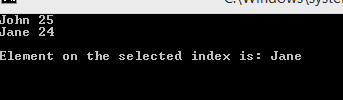
Console.WriteLine();

Console.WriteLine($"Element on the selected index is: {student.Name}");

Console.ReadKey();

}

}



A generic class can have more than one type parameter:

public class CollectionKeyValueInitializer<TKey, TValue>

## Constraints with Generics

Sometimes, we want to ensure that just certain types can be invoked with our generic class. It is often useful while working with classes or interfaces. We can do that by using the where keyword:

public class CollectionInitializer<T> where T: Student

or we can limit our generic class to work only with classes:

public class CollectionInitializer<T> where T: class

There are different variations for this constraints, they depend on the situation we are working in. It is important to know that if we constraint our generic class to work only with classes, we will get an error if we provide value type. If we want to work only with value types, we can constraint our generic class like this:

public class CollectionInitializer<T> where T: struct

## Generic Methods

In the same way that we can create a generic class, we can create a generic method. We just need to set a type parameter in angle brackets right behind a method name:

public void ExampleMethod<T>(T param1, T param2)

{

//Methods body

}

We must pay attention to the type parameter identifier if our generic method exists inside a generic class. If that class has a type T then, our method needs to have a different type (U, Y, R…). Otherwise, the type T from a method will hide the type T from a class.

# Queue, Stack, Hashtable

In this article, we are going to talk about the queue, stack and hash-table collections, how to use them and how to use the methods they provide.

So, let’s start.

## Queue Collection

The queue collection represents a first-in, first-out collection of objects. This means that we can place our objects in a queue collection in a certain order remove those objects by the same order. So, the first object which goes in is the first object to go out.

To create an object instance of a queue collection we can use two different statements.

By using System.Collection.Generic namespace:

Queue<int> intCollection = new Queue<int>();

And by using System.Collection namespace:

Queue queueCollection = new Queue();

If we declare an object by providing a type (in our example an int), we can store only integer numbers inside. On the other hand, if we use the second example we can store different data types in a collection because it stores objects.

## The Most Common Methods and Properties

The Enqueue method adds an element inside a collection:

Queue queueCollection = new Queue();

queueCollection.Enqueue(54);

queueCollection.Enqueue("John");

queueCollection.Enqueue(54.10);

foreach (var item in queueCollection)

{

Console.WriteLine(item);

}

The Dequeue method removes an element at the beginning of the collection and returns it:

Queue queueCollection1 = new Queue();

queueCollection1.Enqueue(54);

queueCollection1.Enqueue("John");

queueCollection1.Enqueue(54.10);

int number = Convert.ToInt32(queueCollection1.Dequeue());

Console.WriteLine($"Removed element is: {number}");

Console.WriteLine();

foreach (var item in queueCollection1)

{

Console.WriteLine(item);

}

The Peek method returns the element at the beginning of the collection but does not remove it:

Queue queueCollection2 = new Queue();

queueCollection2.Enqueue(54);

queueCollection2.Enqueue("John");

queueCollection2.Enqueue(54.10);

int peekNumber = Convert.ToInt32(queueCollection2.Peek());

Console.WriteLine($"Returned element is: {number}");

Console.WriteLine();

foreach (var item in queueCollection2)

{

Console.WriteLine(item);

}

The Clear method removes all the elements from a collection.

If we want to check how many elements we have inside a collection, we can use the Count property:

queueCollection2.Clear();

Console.WriteLine(queueCollection2.Count);

## Stack Collection

The stack collection represents a simple last-in, first-out collection. It means that an element which enters first in a collection will exit last.

As with the Queue collection, we can use the System.Collection and System.Collection.Generic namespaces.

Stack stack = new Stack();

Stack<int> stackInt = new Stack<int>();

## The Most Common Methods and Properties

The Push method inserts an object at the top of the collection:

Stack stack1 = new Stack();

stack1.Push(328);

stack1.Push("Fifty Five");

stack1.Push(124.87);

foreach (var item in stackCollection1)

{

Console.WriteLine(item);

}

The Pop method removes the element which was included last in a collection and returns it:

Stack stackCollection2 = new Stack();

stackCollection2.Push(328);

stackCollection2.Push("Fifty Five");

stackCollection2.Push(124.87);

double number = Convert.ToDouble(stackCollection2.Pop());

Console.WriteLine($"Element removed from a collection is: {number}");

foreach (var item in stackCollection2)

{

Console.WriteLine(item);

}

The Peek method returns an object ready to exit the collection, but it doesn’t remove it:

Stack stackCollection3 = new Stack();

stackCollection3.Push(328);

stackCollection3.Push("Fifty Five");

stackCollection3.Push(124.87);

double number1 = Convert.ToDouble(stackCollection3.Peek());

Console.WriteLine($"Element returned from a collection is: {number}");

foreach (var item in stackCollection3)

{

Console.WriteLine(item);

}

The Clear method removes all objects from a collection.

If we want to count the number of elements, we use the Count property:

stackCollection3.Clear();

Console.WriteLine(stackCollection3.Count);

## Hashtable

The hashtable represents a collection of a key-value pair that is organized based on the hash code of the key. Differently, from the queue and stack collections, we can instantiate a hashtable object by using the only System.Collections namespace:

Hashtable hashTable = new Hashtable();

The hashtable constructor has a fifteen overloaded constructors.

## The Most Common Methods

The Add method adds an element with the specified key and value into the collection:

Hashtable hashTable = new Hashtable();

hashTable.Add(Element.First, 174);

hashTable.Add(Element.Second, "Sixty");

hashTable.Add(Element.Third, 124.24);

foreach (var key in hashTable.Keys)

{

Console.WriteLine($"Key: {key}, value: {hashTable[key]}");

}

The Remove method removes the element with the specified key from a collection:

Hashtable hashTable1 = new Hashtable();

hashTable1.Add(Element.First, 174);

hashTable1.Add(Element.Second, "Sixty");

hashTable1.Add(Element.Third, 124.24);

hashTable1.Remove(Element.Second);

foreach (var key in hashTable1.Keys)

{

Console.WriteLine($"Key: {key}, value: {hashTable[key]}");

}

The ContainsKey method determines whether a collection contains a specific key:

if (hashTable.ContainsKey(Element.Second))

{

Console.WriteLine($"Collection contains key: {Element.Second} and its value is {hashTable[Element.Second]}");

}

The ContainsValue method determines whether a collection contains a specific value.

The Clear method removes all elements from a collection:

hashTable.Clear();

## The Most Common Properties

The Count property counts the number of elements inside a collection:

Console.WriteLine(hashTable.Count);

The Keys property returns all the keys from a collection and the Value property returns all the values from a collection:

Hashtable hashTable2 = new Hashtable();

hashTable2.Add(Element.First, 174);

hashTable2.Add(Element.Second, "Sixty");

hashTable2.Add(Element.Third, 124.24);

var keys = hashTable2.Keys;

foreach (var key in keys)

{

Console.WriteLine(key);

}

Console.WriteLine();

var values = hashTable2.Values;

foreach (var value in values)

{

Console.WriteLine(value);

}

# Generic List and Dictionary

In this article, we are going to talk more about generic collections in C#. A List<T> and Dictionary are very useful collections in C#, and we are going to discover its features in the rest of the article.

## List<T>

A List<T> represents a strongly typed collection of objects that can be accessed by index.

To instantiate a List<T> we need to provide a type between angle brackets:

List<int> numberList = new List<int>();

List<Student> students = new List<Student>();

It has two more constructors that we can use to initialize a List object. With the first one, we can set initial capacity:

List<int> numbers = new List<int>(2);

With the second one, we can populate our list with the IEnumerable colletion:

int[] nums = new int[5] { 1, 2, 3, 4, 5 };

List<int> numbers = new List<int>(nums);

To accees any element we can specify its index position:

int oneNumber = numbers[1];

## Methods and Properties

The Add method adds the element inside a list:

List<int> numbers = new List<int>();

numbers.Add(34);

numbers.Add(58);

numbers.Add(69);

foreach (int number in numbers)

{

Console.WriteLine(number);

}

The AddRange method adds the elements of specified collection to the end of a list:

List<int> numbers = new List<int>();

numbers.Add(34);

numbers.Add(58);

numbers.Add(69);

int[] nums = new int[] { 1, 22, 44 };

numbers.AddRange(nums);

foreach (int number in numbers)

{

Console.WriteLine(number);

}

The Contains method determines whether an element exists in the list:

if(numbers.Contains(34))

{

Console.WriteLine("The number 34 exists in a list");

}

The IndexOf method returns the position of an element as an integer number. If an element couldn’t be found, this method returns -1:

int index;

if((index = numbers.IndexOf(58)) != -1)

{

Console.WriteLine($"The number 58 is on the index: {index}");

}

The LastIndexOf is similar to a previous method except it returns a last occurrence of the element.

The CopyTo method copies the entire collection to a compatible array, starting from the beginning of that array:

int[] copyArray = new int[6];

numbers.CopyTo(copyArray);

foreach (int copyNumber in copyArray)

{

Console.WriteLine(copyNumber);

}

The Remove method removes the first occurrence of a specific element from the list:

numbers.Remove(69);

The Clear method clears all the elements from a list:

numbers.Clear();

We can check how many elements a list has by using the Count property:

Console.WriteLine(numbers.Count);

## Dictionary

Dictionary represents a collection of keys and values. To instantiate an object we can use the following syntax:

Dictionary<KeyType, ValueType> Name = new Dictionary< KeyType, ValueType>();

The KeyType represents a type for our key in a collection. The ValueType represents the value assigned to the key. So we can extract our value from a collection by using the key inside the square brackets:

DictionaryName[key];

Dictionary has several constructors we can use to instantiate objects:

Dictionary<string, int> dictExample = new Dictionary<string, int>();

Dictionary<string, int> dictExample1 = new Dictionary<string, int>(5); //to set initial size

Dictionary<string, int> dictExample2 = new Dictionary<string, int>(dictExample1); //accepts all the elements from created Key-Value collection

## Methods and Properties

The Add method adds the key-value pair inside a collection:

Dictionary<string, int> dictExample = new Dictionary<string, int>();

dictExample.Add("First", 100);

dictExample.Add("Second", 200);

dictExample.Add("Third", 300);

foreach (var item in dictExample)

{

Console.WriteLine(dictExample[item.Key]);

}

The Remove method removes the key-value pair from a collection based on the specified key:

dictExample.Remove("Second");

foreach (var item in dictExample)

{

Console.WriteLine(dictExample[item.Key]);

}

The ContainsKey method determines if a collection contains a specific key.

The ContainsValue method determines if a collection contains a specific value:

if(dictExample.ContainsKey("First"))

{

Console.WriteLine("It contains key");

}

if(dictExample.ContainsValue(300))

{

Console.WriteLine("It contains value");

}

The Clear method removes all key-value pairs from a collection:

dictExample.Clear();

If we want to count all of our elements inside a collection, we can use the Count property. If we want to get a collection of containing Keys or containing Values from a dictionary, we can use the Keys and Values properties:

Console.WriteLine(dictExample.Count);

foreach (var key in dictExample.Keys)

{

Console.WriteLine(key);

}

foreach (var value in dictExample.Values)

{

Console.WriteLine(value);

}

# Delegates

A delegate is a reference to a method. We can use a delegate object to pass it to the code in which we want to call a referenced method, without knowing at compile time which method will be invoked.

A base syntax to create a delegate object is:

delegate Result-Type identifiers([parameters]);

There are three steps in defining and using delegates:

* Declaration of our delegate
* Instantiation, creating the delegate’s object
* Invocation, where we call a referenced method

So let’s see this with an example:

//Declaration

public delegate void WriterDelegate(string text);

class Program

{

public static void Write(string text)

{

Console.WriteLine(text);

}

static void Main(string[] args)

{

//Instantiation

WriterDelegate writerDelegate = new WriterDelegate(Write);

//Invocation

writerDelegate("Some example text.");

}

}

It is important to understand that return type of a method and number of parameters must match a delegates return type and number of parameters. Otherwise, we will get the compiler error. We can see in our example that our Write method has a void as return type and only one string parameter as well as our delegate.

Delegates are very useful in the encapsulation of our methods.

C# has the two built-in delegates: Func<T> and Action<T>, so let’s talk more about them.

## Func<T> Delegate

This delegate encapsulates a method that has up to sixteen parameters and returns a value of the specified type. So, in other words, we use the Func delegate only with a method that has a return type other than void.

We can instantiate the Func delegate with this syntax:

Func<Type1, Type2..., ReturnType> DelegateName = new Func<Type1, Type2..., ReturnType>(MethodName);

We can see that the last parameter inside square brackets is a return type. Of course, we don’t have to initialize a delegate object like this, we can do it in another way:

Func< Type1, Type2..., ReturnType> name = MethodName;

Let’s see how to use Func delegate with an example:

class Program

{

public static int Sum(int a, int b)

{

return a + b;

}

static void Main(string[] args)

{

Func<int, int, int> sumDelegate = Sum;

Console.WriteLine(sumDelegate(10, 20));

}

}

## Action<T> Delegate

This delegate encapsulates a method that has up to sixteen parameters and doesn’t return any result. So we can assign to this delegate only methods with the void return type.

We can instantiate the Action object with this syntax:

Action<Type1, Type2...> DelegateName = new Action<Type1, Type2...>(MethodName);

Or, we can use another way:

Action < Type1, Type2...> DelegateName = MethodName;

Let’s see how to use Func delegate with an example:

public static void Write(string text)

{

Console.WriteLine(text);

}

static void Main(string[] args)

{

Action<string> writeDelegate = Write;

writeDelegate("String parameter to write.");

}

## Practical Example

In this example, we are going to create an application which executes one of three methods (Sum, Subtract, Multiply) based on a single provided parameter. Basically, if we send Sum as a parameter, the Sum method will be executed and so on. First, we will write this example without delegates and then we will refactor that code by introducing delegates.

So let’s start with the first part:

public enum Operation

{

Sum,

Subtract,

Multiply

}

public class OperationManager

{

private int \_first;

private int \_second;

public OperationManager(int first, int second)

{

\_first = first;

\_second = second;

}

private int Sum()

{

return \_first + \_second;

}

private int Subtract()

{

return \_first - \_second;

}

private int Multiply()

{

return \_first \* \_second;

}

public int Execute(Operation operation)

{

switch (operation)

{

case Operation.Sum:

return Sum();

case Operation.Subtract:

return Subtract();

case Operation.Multiply:

return Multiply();

default:

return -1; //just to simulate

}

}

}

class Program

{

static void Main(string[] args)

{

var opManager = new OperationManager(20, 10);

var result = opManager.Execute(Operation.Sum);

Console.WriteLine($"The result of the operation is {result}");

Console.ReadKey();

}

}

If we start this application, we will get the correct response for any operation we send to the Execute method. But this code could be much better and easier to read without switch-case expression. If we have more than ten operations, this switch block would be very ugly to read and maintain as well.

So, let’s change our code to make it readable, maintainable and more object-oriented. Let’s introduce a new class ExecutionManager:

public class ExecutionManager

{

public Dictionary<Operation, Func<int>> FuncExecute { get; set; }

private Func<int> \_sum;

private Func<int> \_subtract;

private Func<int> \_multiply;

public ExecutionManager()

{

FuncExecute = new Dictionary<Operation, Func<int>>(3);

}

public void PopulateFunctions(Func<int> Sum, Func<int> Subtract, Func<int> Multiply)

{

\_sum = Sum;

\_subtract = Subtract;

\_multiply = Multiply;

}

public void PrepareExecution()

{

FuncExecute.Add(Operation.Sum, \_sum);

FuncExecute.Add(Operation.Subtract, \_subtract);

FuncExecute.Add(Operation.Multiply, \_multiply);

}

}

In here, we create a dictionary wich will hold all the operations and all the references towards our methods (Func delegates). Now we can inject this class into the OperationManager class and change the Execute method:

public class OperationManager

{

private int \_first;

private int \_second;

private readonly ExecutionManager \_executionManager;

public OperationManager(int first, int second, ExecutionManager executionManager)

{

\_first = first;

\_second = second;

\_executionManager = executionManager;

\_executionManager.PopulateFunctions(Sum, Subtract, Multiply);

\_executionManager.PrepareExecution();

}

private int Sum()

{

return \_first + \_second;

}

private int Subtract()

{

return \_first - \_second;

}

private int Multiply()

{

return \_first \* \_second;

}

public int Execute(Operation operation)

{

return \_executionManager.FuncExecute.ContainsKey(operation) ?

\_executionManager.FuncExecute[operation]() :

-1;

}

}

Now, we are setting up all in the constructor of the OperationManager class and executing our action in the Execute method if it contains required operation. At the first look, we can see how much better this code is.

Finally, we need to change the Program class:

class Program

{

static void Main(string[] args)

{

var executionManager = new ExecutionManager();

var opManager = new OperationManager(20, 10, executionManager);

var result = opManager.Execute(Operation.Sum);

Console.WriteLine($"The result of the operation is {result}");

Console.ReadKey();

}

}