Object-oriented programming in C#

A Concise Introduction

Adopted by Jes Arbov

This introduction is an adapted version of

C# tutorial

at zetcode.com/lang/csharp

Indhold

Object-oriented programming in C#

There are three widely used programming paradigms: procedural programming, functional programming and object-oriented programming. C# supports both procedural and object-oriented programming.

Object-oriented programming (OOP) is a programming paradigm that uses objects and their interactions to design applications and computer programs.

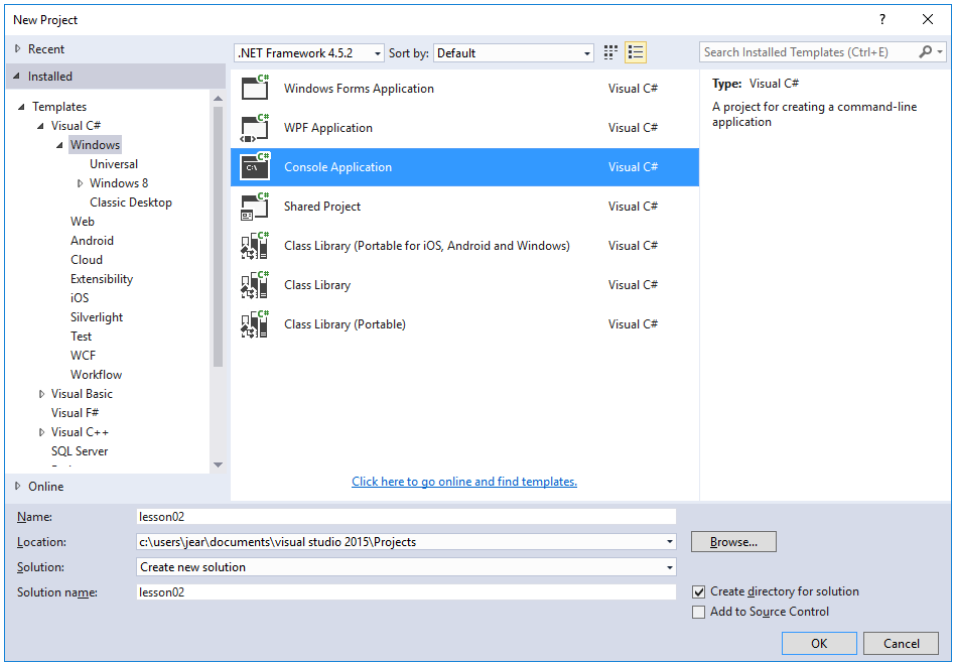
There are some basic programming concepts in OOP:

- Abstraction
- Polymorphism
- Encapsulation
- Inheritance

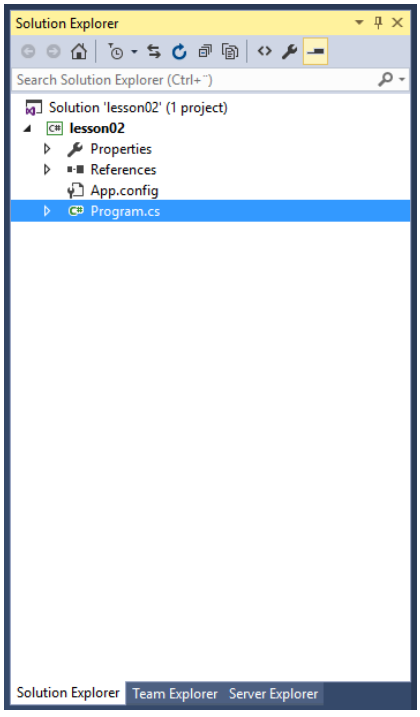
The *abstraction* is simplifying complex reality by modeling classes appropriate to the problem. The *polymorphism* is the process of using an operator or function in different ways for different data input. The *encapsulation* hides the implementation details of a class from other objects. The *inheritance* is a way to form new classes using classes that have already been defined.

In this and the next lesson we'll work with the C# classes. The most efficient way to do that is to use the Visual Studio editor and send the output to the console window. It more efficient than working with ASP.NET MVC because instead of one single class file we had to work with three separate files (class, controller and view files). It makes a lot of sense to do that when you program web applications and want to separate request/response responsibilities (controller) from data (model) and the user interface (view). In this and the next lesson however, we will concentrate on classes and objects exclusively, and for that, a single class file and a simple console window for output is more convenient. In presentations and exercises for both lessons, you will still work within context of the MVC framework.

To follow the examples in this introduction you must first create a new C# Windows Console Application:



Start by deleting the predefined *program.cs* class file:



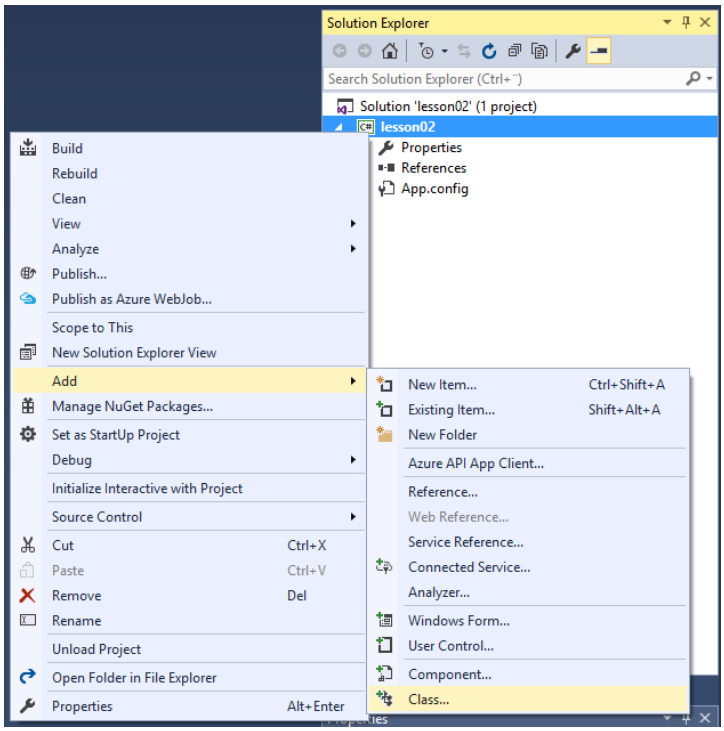
You are now ready to run the following code examples.

Objects

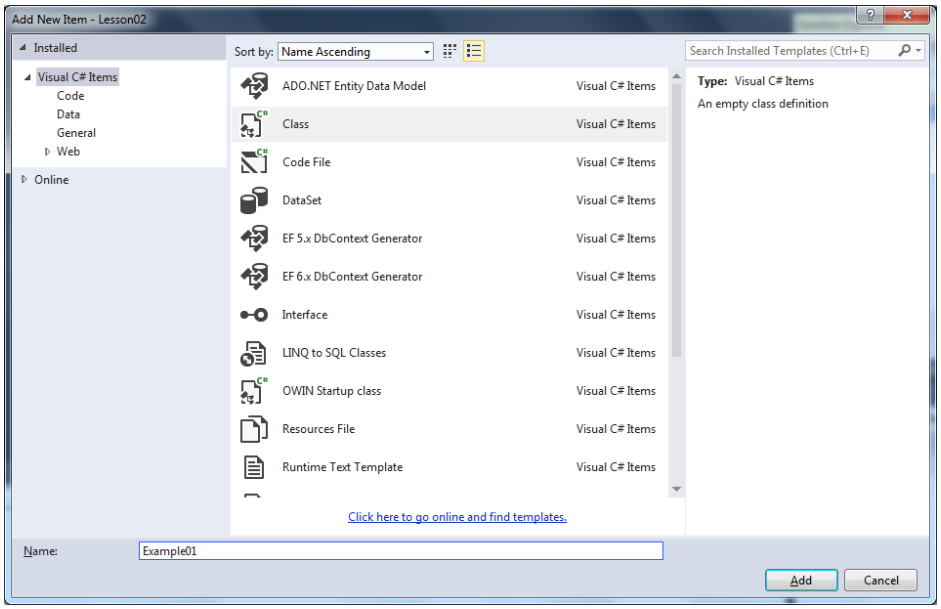
Objects are basic building blocks of a C# OOP program. An object is a combination of data and methods. The data and the methods are called *members* of an object. In an OOP program, we create objects. These objects communicate together through methods. Each object can receive data, return data and process data.

There are two steps in creating an object. First, we define a class. A *class* is a template for an object. It is a blueprint which describes the state and behaviour that the objects of the class all share. A class can be used to create many objects. Objects created at runtime from a class are called *instances* of that particular class.

Create an new class file by right clicking the project name in the **Explorer Window** and choose **Add** -> **Class...**:



name it `example01.cs`:



Delete all pre-inserted code in the `example01.cs` file and substitute it with the code below:

```
using System;

public class EmptyClass {}

public class Objects
{
    static void Main()
    {
        EmptyClass e = new EmptyClass();
        Console.WriteLine(e);
    }
}
```

Example 1

In our first example, we create a simple object.

The first line is a reference to the namespace System. Classes inside the ASP.NET framework is organized in namespaces. By including the System namespace you're able to reference all classes inside that framework from your own user defined classes. Namespaces are like containers used for organizing classes in logical collections.

Tip: Go to http://msdn.microsoft.com/en-us/library/system%28v=vs.110%29.aspx to see a list of the classes organized in the System namespace.

The second line is a declaration of class Being:

```
public class EmptyClass {}
```

This is a class definition in its simplest form. The body of the class declaration, which is what's inside the curly brackets, is totally empty. This class do not have any data or methods.

To create a new instance of a class you must use the `new` keyword:

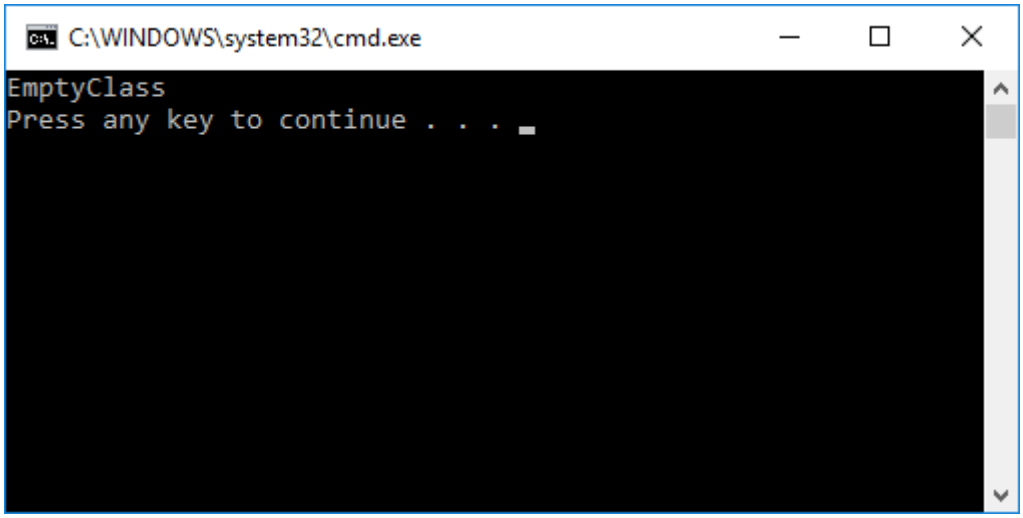
```
EmptyClass e = new EmptyClass();
```

This line creates a new instance of the EmptyClass class. The `e` variable is a reference to the created object.

When we print the object to the console we get a basic description of the object:

```
Console.WriteLine(e);
```

You execute the code and see the output in the console window by pressing Ctrl + F5:



We get the object class name.

What does it mean, to print an object? When we print an object, we in fact call its `ToString()` method. But we have not defined any method yet. It is because every object created inherits from the base `object`. It has some elementary functionality which is shared among all objects created. One of this is the `ToString()` method.

Object fields

Object fields are the data bundled in an instance of a class. The object fields are called *instance variables* or *member fields*. An instance field is a variable defined in a class, for which each object in the class has a separate copy.

```
using System;

public class Person
{
    public string name;
}

public class ObjectFields
{
    static void Main()
    {
        Person p1 = new Person();
        p1.name = "Jane";

        Person p2 = new Person();
        p2.name = "Betty";

        Console.WriteLine(p1.name);
        Console.WriteLine(p2.name);
    }
}
```

Example 2

In the above C# code, we have a Person class with one member field.

```
public class Person
{
    public string name;
}
```

We declare a name member field. The `public` keyword specifies that the member field will be accessible outside the class block.

Tip: In C# the Main method is the entry point of the program. For that reason, each program can only have on entry point. If you try to run a class file inside a C# console project and have two main methods, you'll get an error telling that more than one entry point is defined. To run these examples you should therefore name you main methods

`Main1()`, `Main2()`, `Main3()` etc. and rename the method you want run for the specific example to `Main()`.

When the class is declared we can create an instance of the Person class:

```
Person p1 = new Person();
p1.name = "Jane";
```

After we have create a new instance of Person p1, we set the name variable to "Jane". We use the dot operator to access the fields of objects.

We create another instance of the Person class:

```
Person p2 = new Person();
p2.name = "Betty";
```

Here we set the name variable to "Betty".

```
Console.WriteLine(p1.name);
Console.WriteLine(p2.name);
```

We print the contents of the variables to the Console.

```
Jane
Betty
```

We see the output of the program. Each instance of the Person class has a separate copy of the name member field.

Access modifiers

Access modifiers set the visibility of methods and member fields. C# has four access modifiers: `public`, `protected`, `private` and `internal`. The `public` members can be accessed from anywhere. The `protected` members can be accessed only within the class itself and by inherited and parent classes. The `private` members are limited to the containing type, e.g. only within its class or interface. The `internal` members may be accessed from within the same assembly (exe or dll).

Access modifiers protect data against accidental modifications. They make the programs more robust.

Properties

When you write code in according with principles of object oriented programming you want to follow the principle of *encapsulation*. Encapsulation means that your classes shall provide exactly the information and behavior your code requires and hides all the other details. You classes should be like “black boxes” that hide information that is not relevant for the outside code.

When you follow the principle of encapsulation you protect data in your classes from being accidental changed because only data, which should be allowed to be changed by other classes, is accessible from the outside. You also reduce complexity because the calling code does not need to worry about how the component works or how data are handled internally.

This principle is described in a lot of picturesque ways. For example, you know how to drive a car because you understand its interfaces—the steering wheel and pedals—not because you understand the low-level details about internal combustion and the engine. As a result, you’re able to transfer your knowledge to many different types of automobiles, even if they have dramatically different internal workings.

In accordance with the principle of encapsulation it is recommended best practice not declare field members as public. Instead fields should be accessible though *properties*.

Accessors usually have two parts. The get accessor allows your code to retrieve data from the object. The set accessor allows your code to set the object’s data. In some cases, you might omit one of these parts, such as when you want to create a property that can be examined but not modified.

Property accessors, like any other public piece of a class, should start with an initial capital. This allows you to give the same name to the property accessor and the underlying private variable, because they will have different capitalization, and C# is a case-sensitive

language. (This is one of the rare cases where it's acceptable to differentiate between two elements based on capitalization.) Another option would be to precede the private variable name with an underscore or the characters m_ (for member variable).

```
using System;

public class Person
{
      private string name;
      private int age;

      public string Name
      {
            get
            {
                  return name;
            }
            set
            {
                  name = value; }
            }

      public int Age
      {
            get
            {
                  return age;
            }
            set
            {
                  age = value;
            }
      }
}

public class Properties
{
    static void Main()
    {
        Person p = new Person();
        p.Name = "Jane";

        p.Age = 17;

        Console.WriteLine(p.Name + " is " + p.Age + " years old");
    }
}
```

Example 3

In the above Person class, we have two private member fields and two public properties that are accessible from other classes and is used for reading and setting the values of the belonging member fields.

After we have created a new instance of the `Person` class we can use the dot notation to set the value of the fields by using the properties:

```
Person p = new Person();
p.Name = "Jane";
p.Age = 17;
```

We cannot access the fields directly because they are declared `private`.

Finally, we access both properties to read their values and build a string.

```
Console.WriteLine(p.Name + " is " + p.Age + " years old");
```

Running the example gives this output.

```
Jane is 17 years old
```

Usually, property accessors come in pairs—that is, every property has both a get accessor and a set accessor. But this isn't always the case. You can create properties that can be read but not set (which are called read-only properties) and, less commonly, you can create properties that can be set but not retrieved (called write-only properties). All you need to do is leave out the accessor that you don't need. Here's an example of a *read-only* property:

```
public decimal Price
{
      get {
            return price;
      }
}
```

This technique is particularly handy if you want to create properties that don't correspond directly to a private member variable. For example, you might want to use properties that represent calculated values or use properties that are based on other properties.

You can write as much code as you need inside the get and set code blocks of properties. For example, your property set accessor could raise an error to alert the client code of invalid data and prevent the change from being applied. Or, your property set accessor could change multiple private variables at once, thereby making sure the object's internal state remains consistent. In the Person class example you could ensure that the name was set with in a certain rage:

```
using System;

public class Person
{
      private string name;
      private int age;

      public string Name
      {
            get
            {
                  return name;
            }
            set
            {
                  name = value;
            }
    }

    public int Age
    {
            get
            {
                  return age;}
            set
            {
                  if (value < 12 || value > 120) {
                        throw new Exception("Invalid age");
                  }
                  age = value;
            }
      }
}

public class PropertiesValidation
{
    static void Main()
    {
        Person p = new Person();
        p.Name = "Jane";

        p.Age = 17;

        Console.WriteLine(p.Name + " is " + p.Age + " years old");
    }
}
```

Example 4

Automatic properties

C# version 3.0 introduced auto-implemented or automatic properties. In software applications there are many simple properties that only set or get simple values. To simplify programming and to make the code shorter, automatic properties were created.

```
using System;

public class Pet {
    public string Name { get; set; }
    public int Age { get; set; }
}

public class AutomaticProperties {
    public static void Main() {
        Pet p = new Pet();
        p.Name = "Fritz";
        p.Age = 3;

        Console.WriteLine("{0} is {1} years old",
            p.Name, p.Age);
    }
}
```

Example 5

This code is shorter. We have a person class in which we have two properties.

```
public string Name { get; set; }

public int Age { get; set; }
```

Here we have two automatic properties. There is no implementation of the accessors. And there are no member fields decaled. The compiler does that for us, but the fields are hidden for us and we cannot reference them.

We can reference the properties though the object as usual:

```
Pet p = new Pet();

p.Name = "Fritz";

p.Age = 3;

Console.WriteLine("{0} is {1} years old", p.Name, p.Age);
```

This is the output of the example:

```
Fritz is 4 years old
```

Methods

Objects have *state* and *behavior*. Fields hold state and methods represent the behavior. Methods are functions defined inside the body of a class. They are used to perform operations with the fields of objects. Methods bring *modularity* to our programs.

Methods are essential in the *encapsulation* concept of the OOP paradigm. For example, we might have a Connect() method in our AccessDatabase class. We need not to be informed, how exactly the method Connect() connects to the database. We only have to know, that it is used to connect to a database. This is essential in dividing responsibilities in programming, especially in large applications.

A method is a code block containing a series of statements. Methods must be declared within a class or a structure. It is a good programming practice that methods do only one specific task. Methods bring modularity to programs. Proper use of methods bring the following advantages:

- Reducing duplication of code
- Decomposing complex problems into simpler pieces
- Improving clarity of the code
- Reuse of code
- Information hiding

Basic characteristics of methods are:

- Access level
- Return value type
- Method name
- Method parameters
- Parentheses
- Block of statements

Access level of methods is controlled with access modifiers. They set the visibility of methods. They determine who can call the method. Methods may return a value to the

caller. In case our method returns a value, we provide its data type. If not, we use the `void` keyword to indicate that our method does not return values. Method parameters are surrounded by parentheses and separated by commas. Empty parentheses indicate that the method requires no parameters. The method block is surrounded with { } characters. The block contains one or more statements that are executed, when the method is *invoked*. It is legal to have an empty method block.

A *method signature* is a unique identification of a method for the C# compiler. The signature consists of a method name and the type and kind (value, reference, or output) of each of its formal parameters. Method signature does not include the return type.

You can use any legal character in the name of a method. By convention, method names begin with an uppercase letter. The method names are verbs or verbs followed by adjectives or nouns. Each subsequent word starts with an uppercase character.

We start with a simple example.

```
using System;

public class SimpleClass
{
    public void ShowInfo()
    {
        Console.WriteLine("This is a simple class");
    }
}

public class SimpleMethod
{
    static void Main()
    {
        SimpleClass cs = new SimpleClass ();
        cs.ShowInfo();
    }
}
```

Example 5

We have a `ShowInfo()` method that prints the name of its class.

```
public class SimpleClass
{
    public void ShowInfo()
    {
        Console.WriteLine("This is simple class class");
    }
}
```

Each method must be defined inside a class or a struct. It must have a name. In our case the name is ShowInfo. The keywords that precede the name of the method are access specifier and the return type. Parentheses follow the name of the method. They may contain parameters of the method. Our method does not take any parameters.

```
static void Main()
{
    ...
}
```

This is the Main() method. It is the entry point to each console or GUI application. It must be declared `static`. We will see later why. The return type for a Main() method may be `void` or `int`. The access specifier for the Main() method is omitted. In such a case a default one is used, which is `private`. It is not recommended to use `public` access specifier for the Main() method. It is not supposed to be called by any other methods in the assemblies. It is only the CLR that should be able to call it when the application starts.

```
SimpleClass cs = new SimpleClass();
cs.ShowInfo();
```

We create an instance of the Base class. We call the ShowInfo() method upon the object. We say that the method is an instance method, because it needs an instance to be called. The method is called by specifying the object instance, followed by the member access operator — the dot, followed by the method name.

Method parameters

The method definition specifies the names and types of any parameters that are required. When calling code calls the method, it provides concrete values called arguments for each parameter. The arguments must be compatible with the parameter type but the argument name (if any) used in the calling code does not have to be the same as the parameter named defined in the method.

```
using System;

public class Addition
{
    public int AddTwoValues(int x, int y)
    {
```

```
        return x + y;
    }

    public int AddThreeValues(int x, int y, int z)
    {
        return x + y + z;
    }
}

public class MethodParameters
{
    static void Main()
    {
        Addition a = new Addition();
        int x = a.AddTwoValues(12, 13);
        int y = a.AddThreeValues(12, 13, 14);

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

Example 7

In the above example, we have two methods `AddTwoValues` and `AddThreeValues`. One of them have two parameters, the other three parameters.

```
public int AddTwoValues(int x, int y)
{
    return x + y;
}
```

The AddTwoValues() method have two parameters. These parameters have both `int` as type. The method also returns an integer to the caller. We specify the return type after the access specifier (in this example it is `public`) in the method signature and use the `return` keyword to return a value from the method body.

The AddThreeValues(), which takes three parameters, is similar to the previous method.

```
public int AddThreeValues(int x, int y, int z)
{
    return x + y + z;
}
```

We call the AddTwoValues() method of the addition object:

```
int x = a.AddTwoValues(12, 13);
```

It takes two values. These values are passed as *arguments* to the method. The method returns a value that is assigned to the x variable.

Let's look at another example:

```
using System;

public class Circle
{
    private int radius;


    public int Radius
    {
            set {
                  radius = value;
            }
    }

    public double Area()
    {
        return this.radius * this.radius * Math.PI;
    }
}

public class Methods
{
    static void Main()
    {
        Circle c = new Circle();
        c.Radius = 5;

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

Example 8

In the code example, we have a Circle class. We define a private field, a property and a public method.

The member field is the radius of the circle. The `private` keyword is an access specifier. It tells that the variable is only visible from inside the class. If we want to modify this variable from the outside, we must use the `Radius` property. This way we protect our data.

```
Circle c = new Circle();
c.Radius = 5;
```

We create an instance of the Circle class and set its radius by assigning an integer to the Radius property of the object of the circle. We use the dot operator to call the method.

```
public double Area()
{
    return this.radius * this.radius * Math.PI;
}
```

The `Area()` method returns the area of a circle. The `Math.PI` is a built-in constant.

Running the example gives this outcome:

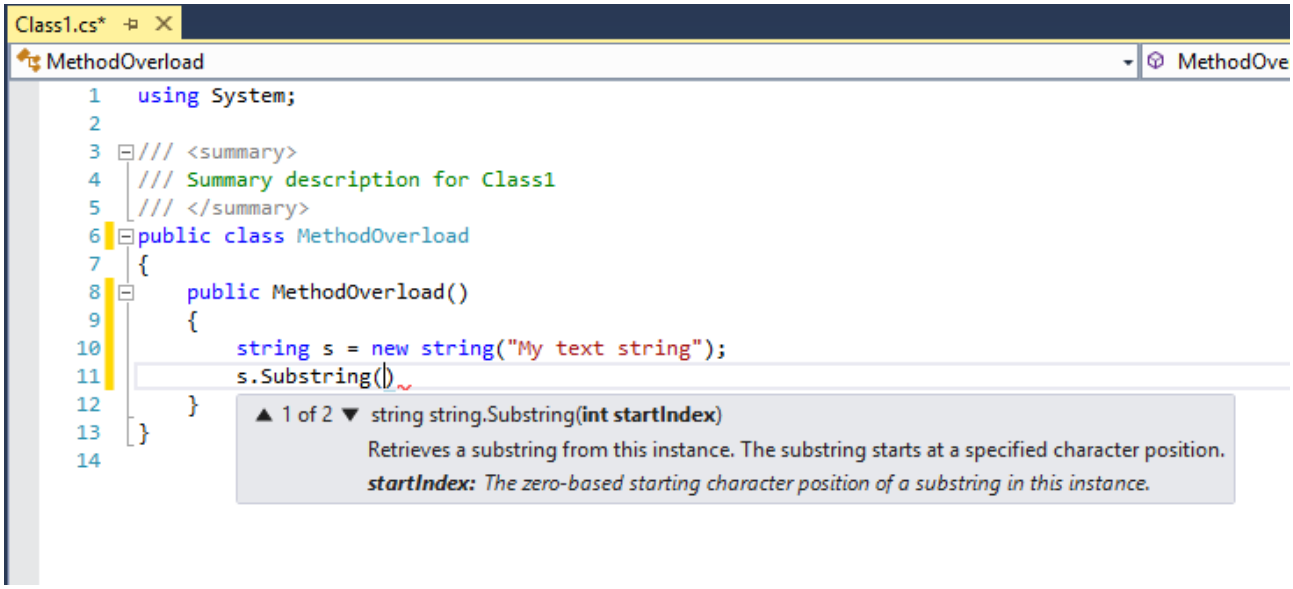
```
78.5398163397448
```

Method overloading

Method overloading allows the creation of several methods with the same name, which differ from each other in the type of the input.

What is method overloading good for? The Qt4 library gives a nice example for the usage. The QPainter class has three methods to draw a rectangle. Their name is drawRect() and their parameters differ. One takes a reference to a floating point rectangle object, another takes a reference to an integer rectangle object, and the last one takes four parameters: x, y, width, height. If the C++ language, which is the language in which Qt is developed, didn't have method overloading, the creators of the library would have to name the methods like drawRectRectF(), drawRectRect(), drawRectXYWH(). The solution with method overloading is more elegant.

The ASP.NET framework uses method overloading intensively. An example is the Substring() method of the string object. As you can see from the grey intellisense helpbox below it as 2 overload methods. The first—which is the one you see—has one parameter, while the second overload the Substring() method has 2 parameters.



Let's take a look at a simple example how to use method overload in practice:

```
using System;

public class Sum
{
    public int GetSum(int x, int y)
    {
        return x + y;
    }

    public int GetSum(int x, int y, int z)
    {
        return x + y + z;
    }
}

public class Overloading
{
    static void Main()
    {
        Sum s = new Sum();

        Console.WriteLine(s.GetSum(30, 40));
        Console.WriteLine(s.GetSum(30, 40, 50));
    }
}
```

Example 9

We have two methods calling GetSum(). They differ in input parameters.

This one takes two parameters:

```
public int GetSum(int x, int y)
{
```

```
    return x + y;
}
```

While this one takes three:

```
public int GetSum(int x, int y, int z)
{
    return x + y + z;
}
```

It's the arguments used for calling the method that decides which version is called:

```
Console.WriteLine(s.GetSum(30, 40));
Console.WriteLine(s.GetSum(30, 40, 50));
```

And this is what we get when we run the example:

```
70
120
```

It is also possible to have more than one method name with the same number of parameters as long as the data types are different:

```
using System;

public class Multiplication
{
    public int GetMultiplication(int x, int y)
    {
        return x * y;
    }

    public double GetMultiplication(double x, double y)
    {
        return x * y;
    }
}

public class OverloadingWithAnEqualNumberOfParameters {
    static void Main() {
        Multiplication m = new Multiplication();

        int a = 3;
        int b = 4;
        double c = 3.3;
        double d = 4.4;

        Console.WriteLine(m.GetMultiplication(a, b));
        Console.WriteLine(m.GetMultiplication(c, d));
    }
}
```

Example 10

The constructor

A constructor is a special kind of a method. It is automatically called when the object is created. Constructors do not return values. The purpose of the constructor is to initiate the state of an object. Constructors have the same name as the class. The constructors are methods, so they can be overloaded too.

If we do not write any constructor for a class, C# provides an implicit default constructor. If we provide a constructor, then a default is not supplied.

```
using System;

public class Student
{
    public Student()
    {
        Console.WriteLine("Student is created");
    }

    public Student (string student)
    {
        Console.WriteLine("Student {0} is created", student);

    }
}

public class Constructor
{
    static void Main()
    {
        new Student();
        new Student("Tom");
    }
}
```

Example 11

We have a Student class. This class has two constructors. The first one does not take parameters; the second one takes one parameter.

This constructor takes one string parameter:

```
public Student(string being)
{
    Console.WriteLine("Student {0} is created", being);
```

```
}
```

Two new instances of the Student class are created. First the constructor without a parameter is called and then the constructor that takes one parameter:

```
new Student();
new Student("Tom");
```

This is the output of the program:

```
Student is created
Student Tom is created
```

In the next example, we initiate two fields (or data members) of the class. Initiation of variables is a typical job for constructors.

```
using System;

public class MyFriend
{
    private DateTime born;
    private string name;

    public MyFriend(string name, DateTime born)
    {
        this.name = name;
        this.born = born;
    }

    public void Info()
    {
        Console.WriteLine("{0} was born on {1}",
            this.name, this.born.ToLongDateString());
    }
}


public class Constructor2
{
    static void Main()
    {
        string name = "Simon";
        DateTime born = new DateTime(1990, 3, 5);

        MyFriend friend = new MyFriend(name, born);
        friend.Info();
    }
}
```

Example 12

We have a MyFriend class with two private fields:

```
private DateTime born;
private string name;
```

In the constructor, we initiate the two fields. The `this` variable is used to reference the object variables which is assigned to the parameter values:

```
public MyFriend(string name, DateTime born)
{
    this.name = name;
    this.born = born;
}
```

We create a MyFriend object by calling the constructor with two arguments. Then we call the `Info()` method of the object.

```
MyFriend fr = new MyFriend(name, born);
fr.Info();
```

The output is:

```
Tom was born on Monday, March 5, 1990
```

Object Initializers

With C# 3.0, initializing both objects and collections have become much easier. Consider this simple Car class, where we use the automatic properties described in a previous chapter:

```
class Car
{
    public string Name { get; set; }
    public string Color { get; set; }
}
```

Now, in C# 2.0, we would have to write a piece of code like this to create a Car instance and set its properties:

```
Car car = new Car();
car.Name = "Chevrolet Corvette";
car.Color = "Yellow”;
```

It's just fine really, but with C# 3.0, it can be done a bit more cleanly, thanks to the new object initializer syntax:

```
Car car = new Car { Name = "Chevrolet Corvette", Color = "Yellow" };
```

As you can see, we use a set of curly brackets after instantiating a new Car object, and within them, we have access to all the public properties of the Car class. This saves a bit of typing, and a bit of space as well. The cool part is that it can be nested too. Consider the following example, where we add a new complex property to the Car class, like this:

```
class Car
{
    public string Name { get; set; }
    public string Color { get; set; }
    public CarManufacturer Manufacturer { get; set; }
}

class CarManufacturer
{
    public string Name { get; set; }
    public string Country { get; set; }
}
```

To initialize a new car with C# 2.0, we would have to do something like this:

```
Car car = new Car();
car.Name = "Corvette";
car.Color = "Yellow";
car.Manufacturer = new CarManufacturer();
car.Manufacturer.Name = "Chevrolet";
car.Manufacturer.Country = "USA";
```

With C# 3.0, we can do it like this instead:

```
Car car = new Car {
                Name = "Chevrolet Corvette",
                Color = "Yellow",
                Manufacturer = new CarManufacturer {
                    Name = "Chevrolet",
                    Country = "USA"
                }
            };
```

Or you could use object initializers within collection initializers:

```
List<Car> cars = new List<Car>
{
    new Car { Name = "Chevrolet Corvette, Color = "Yellow"},
    new Car { Name = "Ruby", Age = 3 }
};
```

Just like with the automatic properties, this is syntactical sugar - you can either use it, or just stick with the old, fashioned way of doing things.

Class constants

C# enables to create class constants. Constants are immutable values which are known at compile time and do not change for the life of the program. Constants are declared with the `const` modifier. These constants do not belong to a concrete object. By convention, constants are written in uppercase letters.

```
using System;

public class Math
{
    public const double PI = 3.14159265359;
}

public class ClassConstants
{
    static void Main()
    {
        Console.WriteLine(Math.PI);
    }
}
```

Example 13

We have a Math class with a PI constant.

```
public const double PI = 3.14159265359;
```

The `const` keyword is used to define a constant. The `public` keyword makes it accessible outside the body of the class. As you can see from the example constants do not belong to the object, they belong to the class. That's the reason why you don't instantiate Math as an object to access the PI constant. Instead, you simply reference the class itself and the constant using the dot notation. If you did try to access PI as a constant though an instance of the Math class you would get an error telling: "'Math.PI' cannot be accessed with an instance reference; qualify it with a type name instead".

Running the example we see this output.

```
3.14159265359
```

Static methods and static member variables

Static methods and static member variables belong to the class (not the object). To call a static member, we use the name of the class and the dot operator followed by the name of the method or member variable. Static methods can only work with static member variables. Static methods are often used to represent data or calculations that do not change in response to object state. An example is a Math library which contains static methods for various calculations (such as PI). We use the `static` keyword to declare a static method. When no static modifier is present, the method is said to be an instance method. We cannot use the `this` keyword in static methods. It can be used in instance methods only.

The Main() method is an entry point to the C# Console and GUI application. In C#, the Main() method is required to be static. Before the application starts, no object is created yet. To invoke non-static methods, we need to have an object instance. Static methods exist before a class is instantiated so static is applied to the main entry point.

```
using System;

public class Basic
{
    static int Id = 2321;

    public static void ShowInfo()
    {
        Console.WriteLine("This is Basic class");
        Console.WriteLine("The Id is: {0}", Id);
    }
}

public class StaticMethod
{
    static void Main()
    {
        Basic.ShowInfo();
    }
}
```

Example 14

In our code example, we define a static ShowInfo() method.

```
static int Id = 2321;
```

A static method can only work with static variables.

```
public static void ShowInfo()
```

```
{
    Console.WriteLine("This is Basic class");
    Console.WriteLine("The Id is: {0}", Id);
}
```

This is our static ShowInfo() method. It works with a static Id member.

To invoke a static method, we do not need an object instance. We call the method by using the name of the class and the dot operator:

```
Basic.ShowInfo();
```

This is the output of the example:

```
This is Basic class
The Id is: 2321
```

Method scope

A variable declared inside a method has a method scope. The *scope* of a name is the region of program text within which it is possible to refer to the entity declared by the name without the qualification of the name. A variable which is declared inside a method has a method scope. It is also called a local scope. The variable is valid only in this particular method.

```
using System;

public class Test
{
    int x = 1;

    public void exec1()
    {
        Console.WriteLine(this.x);
        Console.WriteLine(x);
    }

    public void exec2()
    {
        int z = 5;

        Console.WriteLine(x);
        Console.WriteLine(z);
    }
}

public class MethodScope
{
```

```
    static void Main()
    {
        Test ts = new Test();
        ts.exec1();
        ts.exec2();
    }
}
```

Example 15

In the preceding example, we have the x variable defined outside the exec1() and exec2() methods. The variable has a class scope. It is valid everywhere inside the definition of the Test class, e.g. between its curly brackets.

```
public void exec1()
{
    Console.WriteLine(this.x);
    Console.WriteLine(x);
}
```

The x variable, also called the x field, is an instance variable. And so it is accessible through the `this` keyword. It is also valid inside the exec1() method and can be referred by its bare name. Both statements refer to the same variable.

```
public void exec2()
{
    int z = 5;

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

The x variable can be accessed also in the exec2() method. The z variable is defined in the exec2() method. It has a method scope. It is valid only in this method.

This is the output of the program:

```
1
1
1
5
```

A variable defined inside a method has a local/method scope. If a local variable has the same name as an instance variable, it *shadows* the instance variable. The class variable is still accessible inside the method by using the `this` keyword.

```
using System;

public class Shadow
```

```
{
    int x = 1;

    public void exec()
    {
        int x = 3;

        Console.WriteLine(this.x);
        Console.WriteLine(x);
    }
}

public class Shadowing
{
    static void Main()
    {
        Shadow s = new Test();
        s.exec();
    }
}
```

Example 16

In the preceding example, we declare the x variable outside the exec() method and inside the exec() method. Both variables have the same name, but they are not in conflict because they live in different scopes.

```
Console.WriteLine(this.x);
Console.WriteLine(x);
```

The variables are accessed differently. The x variable defined inside the method, also called the local variable, is simply accessed by its name. The instance variable can be referred by using the `this` keyword.

This is the output of the program:

```
1
3
```

Inheritance

The inheritance is a way to form new classes using classes that have already been defined. The newly formed classes are called *derived* classes, the classes that we derive from are

called *base* classes. Important benefits of inheritance are code reuse and reduction of complexity of a program. The derived classes (descendants) override or extend the functionality of the base classes (ancestors).

```
using System;

public class Creature
{
    public Creature()
    {
        Console.WriteLine("Creature is created");
    }
}

public class Bird : Creature
{
    public Bird()
    {
        Console.WriteLine("Bird is created");
    }
}

public class Inheritance
{
    static void Main()
    {
        new Bird ();
    }
}
```

Example 17

In this program, we have two classes. A base Creature class and a derived Bird class. The derived class inherits from the base class.

```
public class Bird : Being
```

In C#, we use the colon (:) operator to create inheritance relations.

```
new Bird();
```

We instantiate the derived Bird class.

The program outputs:

```
Creature is created
Bird is created
```

We can see that both constructors were called. First, the constructor of the base class is called, then the constructor of the derived class.

A more complex example follows.

```
using System;

public class Being
{
    static int count = 0;

    public Being()
    {
        count++;
        Console.WriteLine("Being is created");
    }

    public void GetCount()
    {
        Console.WriteLine("There are {0} Beings", count);
    }
}

public class Human : Being
{
    public Human()
    {
        Console.WriteLine("Human is created");
    }
}

public class Animal : Being
{
    public Animal()
    {
        Console.WriteLine("Animal is created");
    }
}

public class Dog : Animal
{
    public Dog()
    {
        Console.WriteLine("Dog is created");
    }
}

public class Inheritance2
{
    static void Main()
    {
        new Human();
        Dog dog = new Dog();
        dog.GetCount();
    }
```

```
}
```

Example 18

We have four classes. The inheritance hierarchy is more complicated. The Human and the Animal classes inherit from the Being class. The Dog class inherits directly from the Animal class and indirectly from the Being class. We also introduce a concept of a `static` variable.

```
static int count = 0;
```

We define a `static` variable. Static members are members that are shared by all instances of a class.

```
public Being()
{
    count++;
    Console.WriteLine("Being is created");
}
```

Each time the Being class is instantiated, we increase the count variable by one. This way we keep track of the number of instances created.

```
public class Animal : Being
...

public class Dog : Animal
...
```

The Animal inherits from the Being and the Dog inherits from the Animal. Indirectly, the Dog inherits from the Being as well.

```
new Human();
Dog dog = new Dog();
dog.GetCount();
```

We create instances from the Human and from the Dog classes. We call the `GetCount()` method of the Dog object.

The Human object calls two constructors. The Dog object calls three constructors. There are two Beings instantiated.

```
Being is created
Human is created
Being is created
Animal is created
```

```
Dog is created
There are 2 Beings
```

Sometime, when the base constructor has parameters, we need to call it explicitly from a constructor in a derived class. We use the `base` keyword for that.

```
using System;

public class Shape {
    protected int x;
    protected int y;

    public Shape() {
        Console.WriteLine("Shape is created");
    }

    public Shape(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

public class Cube : Shape {
    private int h;

    public Cube(int x, int y, int h)
        : base(x, y) {
        this.h = h;
    }

    public override string ToString() {
        return String.Format("Cube, x:{1}, y:{2}, h:{0}", x, y, h);
    }
}

public class Shapes {
    static void Main() {
        Cube c = new Cube(2, 5, 6);
        Console.WriteLine(c);
    }
}
```

Example 19

We have two classes. The Shape class and the Cube class. The Shape class is a base class for geometrical shapes. We can put into this class some commonalities of the common shapes, like the x, y coordinates.

```
public Shape()
{
    Console.WriteLine("Shape is created");
```

```
}
public Shape(int x, int y)
{
    this.x = x;
    this.y = y;
}
```

The Shape class has two constructors. The first one is the default constructor. The second one takes two parameters: the x, y coordinates.

```
public Cube(int x, int y, int h) : base(x, y)
{
    this.h = h;
}
```

This is the constructor for the Cube class. This constructor initiates the h member and calls the parent's second constructor, to which it passes the x, y coordinates. Have we not called the constructor explicitly with the `base` keyword, the default constructor of the Shape class would be called.

This is the output of the example:

```
Circle, x:5, y:6, r:2
```

Member fields in derived classes

Member fields with `private` access modifiers are not inherited by derived classes.

```
using System;

public class Base
{
    public string name = "Base";
    protected int id = 5323;
    private bool isDefined = true;
}

public class Derived : Base
{
    public void info()
    {
        Console.WriteLine("This is Derived class");
        Console.WriteLine("Members inherited: ");
        Console.WriteLine(this.name);
        Console.WriteLine(this.id);
        // Console.WriteLine(this.isDefined);
    }
```

```
}

public class CSharpApp
{
    static void Main()
    {
        Derived drv = new Derived();
        drv.info();
    }
}
```

Example 20

In the preceding program, we have a `Derived` class which inherits from the `Base` class. The `Base` class has three member fields. All with different access modifiers. The isDefined member is not inherited. The `private` modifier prevents this.

```
public class Derived : Base
```

The class `Derived` inherits from the `Base` class. To inherit from another class, we use the colon (:) operator.

```
Console.WriteLine(this.name);
Console.WriteLine(this.id);
// Console.WriteLine(this.isDefined);
```

The `public` and the `protected` members are inherited by the Derived class. They can be accessed. The `private` member is not inherited. The line accessing the member field is commented. If we uncommented the line, the code would not compile.

Running the program, we receive this output.

```
This is Derived class
Members inherited:
Base
5323
```

Overriding methods

Now we will introduce two new keywords: the `virtual` keyword and the `override` keyword. They are both method modifiers. They are used to implement polymorphic behaviour of objects. The `virtual` keyword creates a virtual method. Virtual methods can be redefined in derived classes. Later in the derived class we use the `override` keyword to redefine the

method in question. If the method in the derived class is preceded with the `override` keyword, objects of the derived class will call that method rather than the base class method.

```
using System;

public class BaseClass {
    public virtual void Info() {
        Console.WriteLine("This is Base class");
    }
}

public class DerivedClass : BaseClass {
    public override void Info() {
        Console.WriteLine("This is Derived class");
    }
}

public class CSharpAppOverride {
    static void Main() {
        BaseClass[] objs = {new BaseClass(),
            new DerivedClass(), new BaseClass(), new BaseClass(),
            new BaseClass(), new DerivedClass() };

        foreach (BaseClass obj in objs) {
            obj.Info();
        }
    }
}
```

Example 21

We create an array of the BaseClass and DerivedClass objects. We go through the array and invoke the Info() method upon all of them.

This is the virtual method of the BaseClass class. It is expected to be overridden in the derived classes:

```
public virtual void Info()
{
    Console.WriteLine("This is Base class");
}
```

We override the base Info() method in the DerivedClass class. We use the `override` keyword:

```
public override void Info()
{
    Console.WriteLine("This is Derived class");
}
```

Here we create an array of Base and Derived objects. Note that we used the Base type in our array declaration. This is because a Derived class can be converted to the Base class because it inherits from it. The opposite is not true. The only way to have both objects in one array is to use a type which is top most in the inheritance hierarchy for all possible objects.

We traverse the array and call Info() on all objects in the array:

```
BaseClass[] objs = {new BaseClass(), new DerivedClass(),
      new BaseClass (),new BaseClass(), new BaseClass(),
      new DerivedClss() };

foreach (Base obj in objs)
{
    obj.Info();
}
```

This is the output:

```
This is Base class
This is Derived class
This is Base class
This is Base class
This is Base class
This is Derived class
```

Now delete the `override` keyword for `new` keyword. Compile the example again and run it.

This time we have a different output:

```
This is Base class
This is Base class
This is Base class
This is Base class
This is Base class
This is Base class
```

Sealed methods

A sealed method overrides an inherited virtual method with the same signature. A sealed method shall also be marked with the override modifier. Use of the `sealed` modifier prevents a derived class from further overriding the method. The word *further* is

important. First, a method must be virtual. It must be later overridden. And at this point, it can be sealed.

```
using System;

public class A
{
    public virtual void F()
    {
        Console.WriteLine("A.F");
    }

    public virtual void G()
    {
        Console.WriteLine("A.G");
    }
}

public class B : A
{
    public override void F()
    {
        Console.WriteLine("B.F");
    }

    public sealed override void G()
    {
        Console.WriteLine("B.G");
    }
}

public class C : B
{
    public override void F()
    {
        Console.WriteLine("C.F");
    }

    /*public override void G()
    {
        Console.WriteLine("C.G");
    }*/
}


public class SealedMethods
{
    static void Main()
    {
        B b = new B();
        b.F();
        b.G();

        C c = new C();
        c.F();
        c.G();
    }
```

```
}
```

Example 22

In the preceding example, we seal the method G() in class B.

```
public sealed override void G()
{
    Console.WriteLine("B.G");
}
```

The method G() overrides a method with the same name in the ancestor of the B class. It is also sealed to prevent from further overriding the method.

```
/*public override void G()
{
    Console.WriteLine("C.G");
}*/
```

These lines are commented because otherwise the code example would not compile. The Mono compiler would give the following error: sealedmethods.cs(36,26): error CS0239: `C.G()': cannot override inherited member `B.G()' because it is sealed.

```
c.G();
```

This line prints "B.G()" to the Console.

This is the output.

```
B.F
B.G
C.F
B.G
```

Interfaces

A remote control is an interface between the viewer and the TV. It is an interface to this electronic device. Diplomatic protocol guides all activities in the diplomatic field. Rules of the road are rules that motorists, cyclists and pedestrians must follow. Interfaces in programming are analogous to the previous examples.

Objects interact with the outside world with the methods they expose. The actual implementation is not important to the programmer, or it also might be secret. A company might sell a library and it does not want to disclose the actual implementation. A

programmer might call a Maximize() method on a window of a GUI toolkit but knows nothing about how this method is implemented. From this point of view, interfaces are ways through which objects interact with the outside world, without exposing too much about their inner workings.

From the second point of view, interfaces are contracts. If agreed upon, they must be followed. They are used to design an architecture of an application. They help organize the code.

Interfaces are fully abstract types. They are declared using the `interface` keyword. Interfaces can only have signatures of methods, properties, events or indexers. All interface members implicitly have public access. Interface members cannot have access modifiers specified. Interfaces cannot have fully implemented methods, nor member fields. A C# class may implement any number of interfaces. An interface can also extend any number of interfaces. A class that implements an interface must implement all method signatures of an interface.

Interfaces are used to simulate *multiple inheritance*. A C# class can inherit only from one class but it can implement multiple interfaces. Multiple inheritance using the interfaces is not about inheriting methods and variables. It is about inheriting ideas or contracts, which are described by the interfaces.

There is one important distinction between interfaces and abstract classes. Abstract classes provide partial implementation for classes that are related in the inheritance hierarchy. Interfaces on the other hand can be implemented by classes that are not related to each other. For example, we have two buttons. A classic button and a round button. Both inherit from an abstract button class that provides some common functionality to all buttons. Implementing classes are related, since all are buttons. Another example might have classes Database and SignIn. They are not related to each other. We can apply an ILoggable interface that would force them to create a method to do logging.

```
using System;

public interface IInfo
{
    void DoInform();
}

public class Some : IInfo
```

```
{
    public void DoInform()
    {
        Console.WriteLine("This is Some Class");
    }
}

public class SimpleInterface
{
    static void Main()
    {
        Some sm = new Some();
        sm.DoInform();
    }
}
```

Example 23

This is a simple C# program demonstrating an interface.

```
public interface IInfo
{
    void DoInform();
}
```

This is an interface `IInfo`. It has the `DoInform()` method signature.

```
public class Some : IInfo
```

We implement the `IInfo` interface. To implement a specific interface, we use the colon (:) operator.

```
public void DoInform()
{
    Console.WriteLine("This is Some Class");
}
```

The class provides an implementation for the `DoInform()` method.

The next example shows how a class can implement multiple interfaces.

```
using System;

public interface Device
{
    void SwitchOn();
    void SwitchOff();
}

public interface Volume
{
    void VolumeUp();
```

```
    void VolumeDown();
}

public interface Pluggable
{
    void PlugIn();
    void PlugOff();
}

public class CellPhone : Device, Volume, Pluggable
{
    public void SwitchOn()
    {
        Console.WriteLine("Switching on");
    }

    public void SwitchOff()
    {
        Console.WriteLine("Switching on");
    }

    public void VolumeUp()
    {
        Console.WriteLine("Volume up");
    }

    public void VolumeDown()
    {
        Console.WriteLine("Volume down");
    }

    public void PlugIn()
    {
        Console.WriteLine("Plugging In");
    }

    public void PlugOff()
    {
        Console.WriteLine("Plugging Off");
    }
}

public class MultipleInterfaces
{
    static void Main()
    {
        CellPhone cp = new CellPhone();

        cp.SwitchOn();
        cp.VolumeUp();
        cp.PlugIn();
    }
}
```

Example 24

We have a `CellPhone` class that inherits from three interfaces.

```
public class CellPhone : Device, Volume, Pluggable
```

The class implements all three interfaces, which are divided by a comma. The CellPhone class must implement all method signatures from all three interfaces.

Running the program we get this output:

```
Switching on
Volume up
Plugging In
```

The next example shows how interfaces can inherit from multiple other interfaces.

```
using System;

public interface IInform {
    void DoInform();
}

public interface IVers {
    void GetVersion();
}

public interface ISysLog : IInform, IVers {
    void DoLog();
}

public class DBConnect : ISysLog {

    public void DoInform() {
        Console.WriteLine("This is DBConnection class");
    }

    public void GetVersion() {
        Console.WriteLine("Version 1.02");
    }

    public void DoLog() {
        Console.WriteLine("Logging");
    }

    public void Connect() {
        Console.WriteLine("Connecting to the database");
    }
}

public class InterfaceHierarchy {
    static void Main() {
        DBConnect db = new DBConnect();

        db.DoInform();
        db.GetVersion();
        db.DoLog();
        db.Connect();
    }
```

```
}
```

Example 25

We define three interfaces. We can organize interfaces in a hierarchy.

```
public interface ISysLog : IInform, IVers
```

The `ISysLog` interface inherits from two other interfaces.

```
public void DoInform()
{
    Console.WriteLine("This is DBConnection class");
}
```

The `DBConnection` class implements the `DoInform()` method. This method was inherited by the `ISysLog` interface, which the class implements.

This is the output:

```
This is DBConnection class
Version 1.02
Logging
Connecting to the database
```

Abstract classes and methods

Abstract classes cannot be instantiated. If a class contains at least one abstract method, it must be declared abstract too. Abstract methods cannot be implemented; they merely declare the methods' signatures. When we inherit from an abstract class, all abstract methods must be implemented by the derived class. Furthermore, these methods must be declared with the same of less restricted visibility.

Unlike *Interfaces*, abstract classes may have methods with full implementation and may also have defined member fields. So abstract classes may provide a partial implementation. Programmers often put some common functionality into abstract classes. And these abstract classes are later subclassed to provide more specific implementation. For example, the Qt graphics library has a QAbstractButton, which is the abstract base class of button widgets, providing functionality common to buttons. Buttons Q3Button,

QCheckBox, QPushButton, QRadioButton, and QToolButton inherit from this base abstract class.

Formally put, abstract classes are used to enforce a protocol. A protocol is a set of operations which all implementing objects must support.

```
using System;

public abstract class Drawing
{
    protected int x = 0;
    protected int y = 0;

    public abstract double Area();

    public string GetCoordinates()
    {
        return string.Format("x: {0}, y: {1}", this.x, this.y);
    }
}

public class Circle : Drawing
{
    private int r;

    public Pie(int x, int y, int r)
    {
        this.x = x;
        this.y = y;
        this.r = r;
    }

    public override double Area()
    {
        return this.r * this.r * Math.PI;
    }

    public override string ToString()
    {
        return string.Format("Circle at x: {0}, y: {1}, radius: {2}",
            this.x, this.y, this.r);
    }
}

public class AbstractClass
{
    static void Main()
    {
        Pie p = new Pie (12, 45, 22);
        Console.WriteLine(p);
        Console.WriteLine("Area of pie: {0}", p.Area());
        Console.WriteLine(p.GetCoordinates());
    }
}
```

Example 26

We have an abstract base Drawing class. The class defines two member fields, defines one method and declares one method. One of the methods is abstract, the other one is fully implemented. The Drawing class is abstract because we cannot draw it. We can draw a circle, a dot or a square. The Drawing class has some common functionality to the objects that we can draw.

```
public abstract class Drawing
```

We use the `abstract` keyword to define an abstract class.

```
public abstract double Area();
```

An abstract method is also preceded with the `abstract` keyword.

```
public class Pie : Drawing
```

A Pie is a subclass of the Drawing class. It must implement the abstract `Area()` method.

```
public override double Area()
{
    return this.r * this.r * Math.PI;
}
```

When we implement the `Area()` method, we must use the `override` keyword. This way we inform the compiler that we override an existing (inherited) method.

We see the output of the program:

```
Pie at x: 12, y: 45, radius: 22
Area of pie: 1520.53084433746
x: 12, y: 45
```

Sealed classes

The `sealed` keyword is used to prevent unintended derivation from a class. A sealed class cannot be an abstract class.

```
using System;

sealed class Math
```

```
{
    public static double GetPI()
    {
        return 3.141592;
    }
}

public class DerivedMath : Math
{

    public void Say()
    {
        Console.WriteLine("Derived class");
    }
}

public class CSharpApp
{
    public static void Main()
    {
        DerivedMath dm = new DerivedMath();
        dm.Say();
    }
}
```

In the above program, we have a base Math class. The sole purpose of this class is to provide some helpful methods and constants to the programmer. (In our case we have only one method for simplicity reasons.) It is not created to be inherited from. To prevent uninformed other programmers to derive from this class, the creators made the class `sealed`. If you try to compile this program, you get the following error: 'DerivedMath' cannot derive from sealed class `Math'.

Polymorphism

The polymorphism is the process of using an operator or function in different ways for different data input. In practical terms, polymorphism means that if class B inherits from class A, it does not have to inherit everything about class A; it can do some of the things that class A does differently.

In general, polymorphism is the ability to appear in different forms. Technically, it is the ability to redefine methods for derived classes. Polymorphism is concerned with the application of specific implementations to an interface or a more generic base class.

Polymorphism is the ability to redefine methods for derived classes.

```
using System;

public abstract class Quadrilateral
{
    protected int x;
    protected int y;

    public abstract int Area();
}

public class Rectangle : Quadrilateral
{
    public Rectangle(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public override int Area()
    {
        return this.x * this.y;
    }
}

public class Square : Quadrilateral
{
    public Square(int x)
    {
        this.x = x;
    }

    public override int Area()
    {
        return this.x * this.x;
    }
}

public class Polymorphism
{
    static void Main()
    {
        Quadrilateral[] quadrilaterals = { new Square(5),
            new Rectangle(9, 4), new Square(12) };

        foreach (Quadrilateral quadrilateral in quadrilaterals)
        {
            Console.WriteLine(quadrilateral.Area());
        }
    }
}
```

Example 27

In the above program, we have an abstract `Quadrilateral` class. This class morphs into two descendant classes: `Rectangle` and `Square`. Both provide their own implementation of the `Area()` method. Polymorphism brings flexibility and scalability to the OOP systems.

```
public override int Area()
{
    return this.x * this.y;
}
...
public override int Area()
{
    return this.x * this.x;
}
```

The `Rectangle` and the `Square` classes have their own implementations of the `Area()` method.

```
Quadrilateral[] quadrilaterals = { new Square(5),
    new Rectangle(9, 4), new Square(12) };
```

We create an array of three Quadrilaterals.

```
foreach (Quadrilateral quadrilateral in quadrilaterals)
{
    Console.WriteLine(quadrilateral.Area());
}
```

We go through each quadrilateral and call the `Area()` method on it. The compiler calls the correct method for each Quadrilateral. This is the essence of polymorphism.

The program prints:

```
25
36
144
```

Exceptions

Exceptions are designed to handle the occurrence of exceptions, special conditions that change the normal flow of program execution. Exceptions are raised or thrown.

During the execution of our application many things might go wrong. A disk might get full and we cannot save our file. An Internet connection might go down while our application tries to connect to a site. All these might result in a crash of our application. It is a responsibility of a programmer to handle errors that can be anticipated.

The `try`, `catch` and `finally` keywords are used to work with exceptions.

```
using System;

public class DivisionByZero
{
    static void Main()
    {
        int x = 100;
        int y = 0;
        int z;

        try
        {
            z = x / y;

        } catch (ArithmeticException e)
        {
            Console.WriteLine("An exception occured");
            Console.WriteLine(e.Message);
        }
    }
}
```

Example 28

In the above program, we intentionally divide a number by zero. This leads to an error. Statements that are error prone are placed in the `try` block:

```
try
{
    z = x / y;
```

Exception types follow the `catch` keyword. In our case we have an `ArithmeticException`.

```
}
} catch (ArithmeticException e)
{
    Console.WriteLine("An exception occured");
    Console.WriteLine(e.Message);
}
```

This exception is thrown for errors in an arithmetic, casting, or conversion operation. Statements that follow the `catch` keyword are executed when an error occurs. When an exception occurs, an exception object is created. From this object we get the `Message` property and print it to the Console.

Output of the code example:

```
An exception occured
Attempted to divide by zero . . .
```

Any uncaught exception in the current context propagates to a higher context and looks for an appropriate catch block to handle it. If it can't find any suitable catch blocks, the default mechanism of the .NET runtime will terminate the execution of the entire program.

```
using System;

public class UncaughtException
{
    static void Main()
    {
        int x = 100;
        int y = 0;

        int z = x / y;

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

Example 29

In this program, we divide by zero. There is no custom exception handling.

The C# compiler gives this error message:

```
Unhandled Exception: System.DivideByZeroException: Attempted to
divide by zero at UncaughtException.Main() in [file line number]
```

In the following example, we will read the contents of a file.

```
using System;
using System.IO;

public class ReadFile
{
    static void Main()
    {
        FileStream fs = new FileStream("langs", FileMode.OpenOrCreate);

        try
        {
            StreamReader sr = new StreamReader(fs);
            string line;

            while ((line = sr.ReadLine()) != null)
            {
                Console.WriteLine(line);
            }

        } catch (IOException e)
        {
            Console.WriteLine("IO Error");
            Console.WriteLine(e.Message);
```

```
        } finally
        {
            Console.WriteLine("Inside finally block");

            if (fs != null)
                fs.Close();
        }
    }
}
```

Example 30

The statements following the `finally` keyword are always executed. It is often used for clean-up tasks, such as closing files or clearing buffers.

```
} catch (IOException e)
{
    Console.WriteLine("IO Error");
    Console.WriteLine(e.Message);

}
```

In this case, we catch for a specific `IOException` exception.

```
} finally
{
    Console.WriteLine("Inside finally block");

    if (fs != null)
        fs.Close();
}
```

These lines guarantee that the file handler is closed.

We show the contents of the langs file with the cat command and output of the program:

```
C#
Python
C++
Java
```

We often need to deal with multiple exceptions.

```
using System;
using System.IO;

public class MultipleExceptions
{
    static void Main()
    {
        int x;
        int y;
        double z;

        try
```

```
        {
            Console.Write("Enter first number: ");
            x = Convert.ToInt32(Console.ReadLine());

            Console.Write("Enter second number: ");
            y = Convert.ToInt32(Console.ReadLine());

            z = x / y;
            Console.WriteLine("Result: {0:N} / {1:N} = {2:N}", x, y, z);

        } catch (DivideByZeroException e)
        {
            Console.WriteLine("Cannot divide by zero");
            Console.WriteLine(e.Message);

        } catch (FormatException e)
        {
            Console.WriteLine("Wrong format of number.");
            Console.WriteLine(e.Message);
        }
    }
}
```

Example 31

In this example, we catch for various exceptions. Note that more specific exceptions should precede the generic ones. We read two numbers from the Console and check for zero division error and for wrong format of number.

Running the example we get this outcome:

```
Enter first number: we
Wrong format of number.
Input string was not in the correct format
```

Custom exceptions are user defined exception classes that derive from the `System.Exception` class.

```
using System;

class BigValueException : Exception
{
    public BigValueException(string msg) : base(msg) {}
}

public class CustomException
{
    static void Main()
    {
        int x = 340004;
        const int LIMIT = 333;

        try
        {
            if (x > LIMIT)
            {
```

```
                throw new BigValueException("Exceeded the maximum value");
            }

        } catch (BigValueException e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

Example 32

We assume that we have a situation in which we cannot deal with big numbers.

```
class BigValueException : Exception
```

We have a BigValueException class. This class derives from the built-in `Exception` class.

```
const int LIMIT = 333;
```

Numbers bigger than this constant are considered to be "big" by our program.

```
public BigValueException(string msg) : base(msg) {}
```

Inside the constructor, we call the parent's constructor. We pass the message to the parent.

```
if (x > LIMIT)
{
    throw new BigValueException("Exceeded the maximum value");
}
```

If the value is bigger than the limit, we throw our custom exception. We give the exception a message "Exceeded the maximum value".

We catch the exception and print its message to the Console:

```
} catch (BigValueException e)
{
    Console.WriteLine(e.Message);
}
```

This is the output of the program:

```
Exceeded the maximum value
```

Namespaces

Namespaces are C# program elements designed to help you organize your code. They also provide assistance in avoiding name clashes between two sets of code. Implementing Namespaces in your own code is a good habit because it is likely to save you from problems later when you want to reuse some of your code. For example, if you created a class named Console, you would need to put it in your own namespace to ensure that there wasn't any confusion about when the System.Console class should be used or when your class should be used. Generally, it would be a bad idea to create a class named Console, but in many cases your classes will be named the same as classes in either the .NET Framework Class Library or a third party library and namespaces help you avoid the problems that identical class names would cause.

Namespaces do not correspond to file or directory names. If naming directories and files to correspond to namespaces helps you organize your code, then you may do so, but it is not required.

Let's look at a simple namespace declaration:

```
// Namespace Declaration
using System;

// The MyCompany Namespace
namespace MyCompany
{
    // Program start class
    class NamespaceMyCompany
    {
        // Main begins program execution.
        public static void Main()
        {
            // Write to console
            Console.WriteLine("This is the new MyCompany Namespace.");
        }
    }
}
```

Example 33

The example shows how to create a namespace. We declare the new namespace by putting the word namespace in front of MyCompany. Curly braces surround the members inside the MyCompany namespace.

You can nest namespaces:

```
// Namespace Declaration
using System;

// The MyCompany Project1 Namespace
namespace MyCompany
{
    namespace Project1
    {
        // Program start class
        class NestedNamespaces
        {
            // Main begins program execution.
            public static void Main()
            {
                // Write to console
                Console.WriteLine("This is the new MyCompany Project
Namespace.");
            }
        }
    }
}
```

Example 34

Namespaces allow you to create a system to organize your code. A good way to organize your namespaces is via a hierarchical system. You put the more general names at the top of the hierarchy and get more specific as you go down. This hierarchical system can be represented by nested namespaces. The example shows how to create a nested namespace. By placing code in different sub-namespaces, you can keep your code organized.

There is an alternative syntax for nesting:

```
// Namespace Declaration
using System;

// The C# Station Tutorial Namespace
namespace MyCompany.Project1
{
    // Program start class
    class NestedNamespacesAltSyntax
    {
        // Main begins program execution.
        public static void Main()
        {
            // Write to console
            Console.WriteLine("This is the new MyCompany Project1 Namespace.");
        }
    }
}
```

Example 35

The example shows a nested namespace specified with the dot operator between MyCompany and Project1. The result is the same as the former example. However, it is easier to write.

This example shows how to call a class inside another namespace:

```
// Namespace Declaration
using System;

namespace MyCompany
{
    // nested namespace
    namespace Project1
    {
        class MyExample
        {
            public static void MyPrint()
            {
                Console.WriteLine("First example of calling another namespace
member.");
            }
        }
    }

    // Program start class
    class NamespaceCalling
    {
        // Main begins program execution.
        public static void Main()
        {
            // Write to console
            Project1.MyExample.MyPrint();
            MyCompany.Project2.MyExample.MyPrint();
        }
    }
}

// same namespace as nested namespace above
namespace MyCompany.Project2
{
    class MyExample
    {
        public static void MyPrint()
        {
            Console.WriteLine("Second example of calling another namespace
member.");
        }
    }
}
```

Example 36

The example demonstrates how to call namespace members with fully qualified names. A fully qualified name contains every language element from the namespace name down to the method call. At the top of the listing there is a nested namespace Project1 within the MyCompany namespace with class MyExample and method MyPrint. Main() calls this method with the fully qualified name of Project1.MyExample.MyPrint(). Since Main() and

the tutorial namespace are located in the same namespace, using MyCompany.Project1 is unnecessary.

At the bottom of the example, there is an additional MyCompany.Project2 namespace. The classes MyCompany.Project1.MyExample and MyCompany.Project2.MyExample belong to different namespaces. Because of that, we can give the classes the same name and they can have methods with the same name. Because they are in different namespaces we avoid a name clash between them. In the Main() method the MyPrint() method of both classes is called with the fully qualified name Project1.MyExample.MyPrint() and MyCompany.Project2.MyExample.MyPrint().

If you would like to call methods without typing their fully qualified name, you can implement the using directive.

```
// Namespace Declaration
using System;
using MyCompany.Project1;


// MyCompay Project1 Namespace
namespace MyCompany.Project1
{
    class MyExample2
    {
        public static void MyPrint()
        {
            Console.WriteLine("Example of using a using directive.");
        }
    }
}

// Program start class
class UsingDirective
{
    // Main begins program execution.
    public static void Main()
    {
        // Call namespace member
        MyExample2.MyPrint();
    }
}
```

Example 37

The above example shows two using directives. The first, using System, is the same using directive you have seen in every program in this tutorial. It allows you to type the method names of members of the System namespace without typing the word System every time. Console is a class member of the System namespace. The fully qualified name to its method WriteLine() is System.Console.WriteLine().

Similarly, the using directive using MyCompany.Project1 allows us to call members of the MyCompany.Project1 namespace without typing the fully qualified name. This is why we can type MyExample2.myPrint(). Without the using directive, we would have to type MyCompany.Project1.myExample.myPrint() every time we wanted to call that method.

In these examples, namespaces have contained classes and other namespaces. However, namespaces can hold other types. Within a namespace, you can declare one or more of the following types:

- another namespace
- class
- interface
- struct
- enum
- delegate

Whether or not you explicitly declare a namespace in a C# source file, the compiler adds a default namespace. This unnamed namespace, sometimes referred to as the global namespace, is present in every file. Any identifier in the global namespace is available for use in a named namespace.

Resources

C# tutorial
http://zetcode.com/lang/csharp/

C# Tutorial
http://www.tutorialspoint.com/csharp/index.htm

The C# Station Tutorial
http://www.csharp-station.com/Tutorial/CSharp/