OOP – Object Oriented Programming

Learning Outcomes

At the end of the topic, the students must have:

1. Developed a program using C# methods
2. Learned how to use parameters and arguments
3. Differentiate objects and classes in OOP
4. Familiarized about the Object-Oriented Programming

Content

Method

A method makes a large section of code into smaller. It is re-usable parts that make the program easier to understand.

A method is a block of code which only runs when it is called. You can pass data, known as parameters, into a method. Methods are used to perform certain actions, and they are also known as functions. Why use methods? To reuse code: define the code once, and use it many times.

To use a method, you need to −

- Define the method
- Call the method

Defining Methods in C#

When you define a method, you basically declare the elements of its structure. The syntax for defining a method in C# is as follows –

```
<Access Specifier> <Return Type> <Method Name>(Parameter List) {
   Method Body
}
```

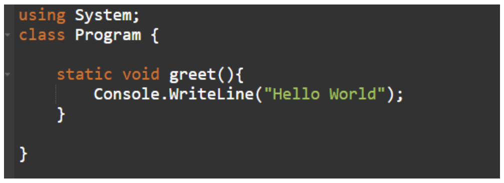
Following are the various elements of a method –

- **Access Specifier** - This determines the visibility of a variable or a method from another class.
- **Type (return type)** - A method may return a value. The return type is the data type of the value the method returns. If the method is not returning any values, then the return type is void.

- **Function/method Name** - Method name is a unique identifier and it is case sensitive. It cannot be same as any other identifier declared in the class.
- **Parameter/s** - Enclosed between parentheses, the parameters are used to pass and receive data from a method. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.
- **Method Body** - This contains the set of instructions needed to complete the required activity.

Creating a Method

Create a method inside the Program class:

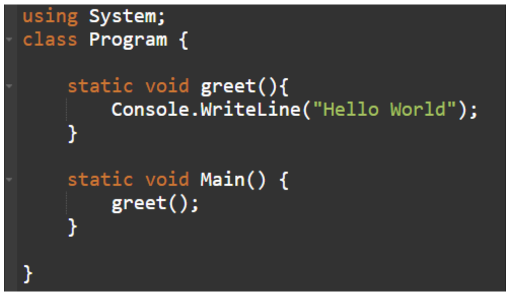


```
using System;
class Program {

    static void greet(){
        Console.WriteLine("Hello World");
    }

}
```

Call a Method

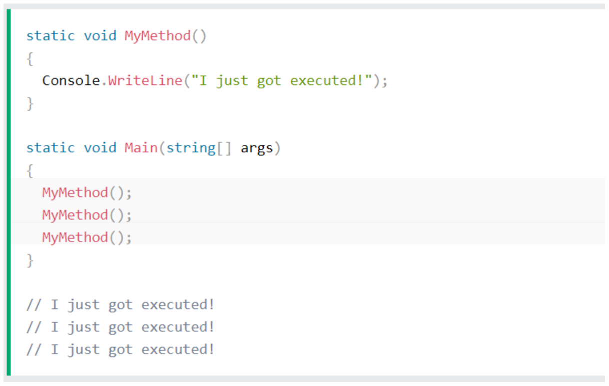
To call (execute) a method, write the method's name followed by two parentheses () and a semicolon;

Example: Inside Main(), call the greet() method:



```
using System;
class Program {

    static void greet(){
        Console.WriteLine("Hello World");
    }

    static void Main() {
        greet();
    }

}
```

A method can be called multiple times:



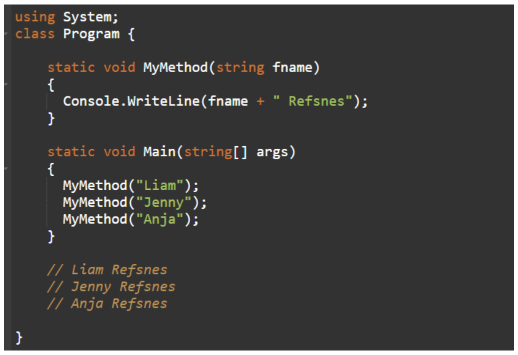
```
static void MyMethod()
{
  Console.WriteLine("I just got executed!");
}

static void Main(string[] args)
{
  MyMethod();
  MyMethod();
  MyMethod();
}

// I just got executed!
// I just got executed!
// I just got executed!
```

Parameters and Arguments

Information can be passed to methods as parameter. Parameters act as variables inside the method.

They are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a method that takes a string called fname as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

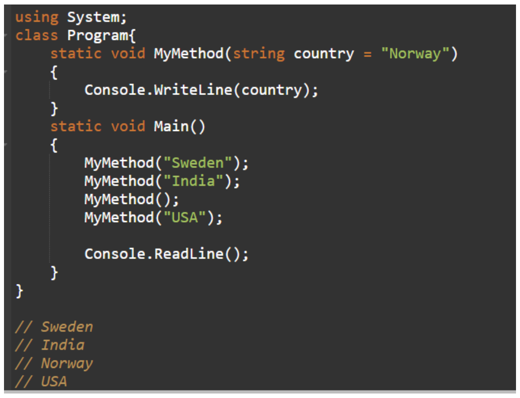


```
using System;
class Program {

    static void MyMethod(string fname)
    {
      Console.WriteLine(fname + " Refsnes");
    }

    static void Main(string[] args)
    {
      MyMethod("Liam");
      MyMethod("Jenny");
      MyMethod("Anja");
    }

    // Liam Refsnes
    // Jenny Refsnes
    // Anja Refsnes

}
```

When a parameter is passed to the method, it is called an argument. So, from the example above: *fname* is a parameter, while *Liam*, *Jenny* and *Anja* are arguments.

Default Parameter Value

You can also use a default parameter value, by using the equals sign (=). If we call the method without an argument, it uses the default value ("Norway"):

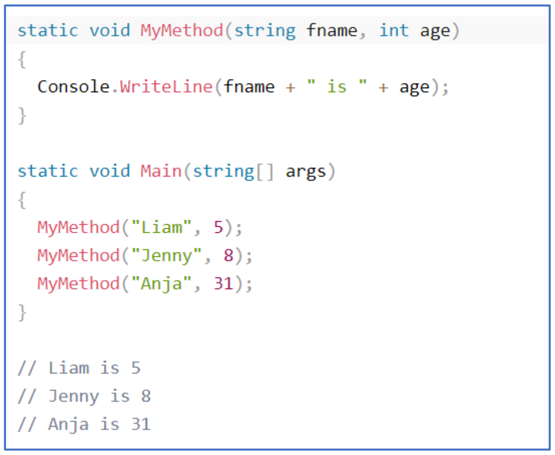


```
using System;
class Program{
    static void MyMethod(string country = "Norway")
    {
        Console.WriteLine(country);
    }
    static void Main()
    {
        MyMethod("Sweden");
        MyMethod("India");
        MyMethod();
        MyMethod("USA");

        Console.ReadLine();
    }
}

// Sweden
// India
// Norway
// USA
```

A parameter with a default value, is often known as an "optional parameter". From the example above, *country* is an optional parameter and *"Norway"* is the default value.

Multiple Parameters

You can have as many parameters as you like:

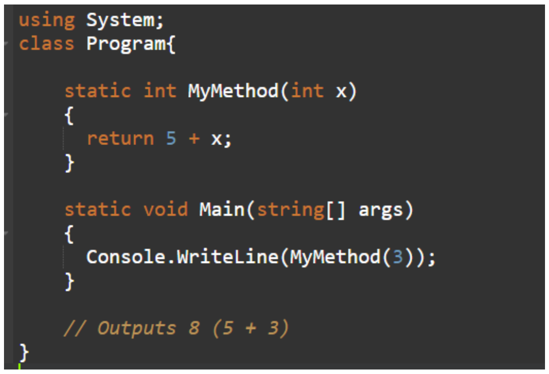


```
static void MyMethod(string fname, int age)
{
  Console.WriteLine(fname + " is " + age);
}

static void Main(string[] args)
{
  MyMethod("Liam", 5);
  MyMethod("Jenny", 8);
  MyMethod("Anja", 31);
}

// Liam is 5
// Jenny is 8
// Anja is 31
```

Note that when you are working with multiple parameters, the method call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

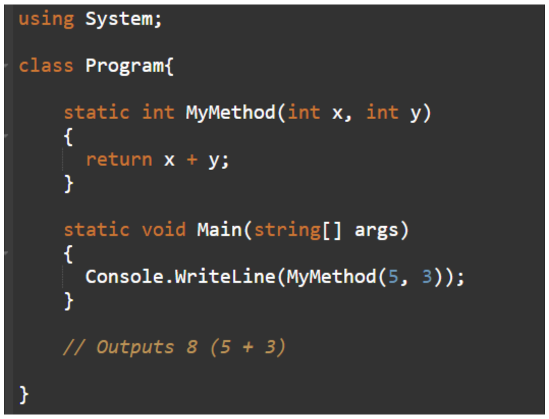
Return Values

The void keyword, used in the examples above, indicates that the method should not return a value. If you want the method to return a value, you can use a primitive data type (such as int or double) instead of void, and use the return keyword inside the method:



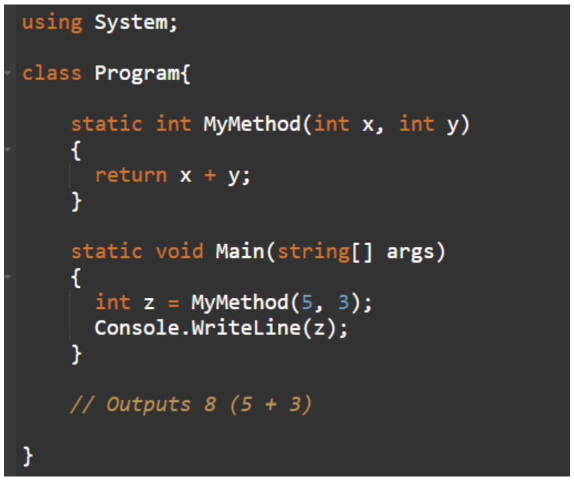
```
using System;
class Program{

    static int MyMethod(int x)
    {
      return 5 + x;
    }

    static void Main(string[] args)
    {
      Console.WriteLine(MyMethod(3));
    }

    // Outputs 8 (5 + 3)
}
```

This example returns the sum of a method's **two parameters**:



```
using System;

class Program{

    static int MyMethod(int x, int y)
    {
      return x + y;
    }

    static void Main(string[] args)
    {
      Console.WriteLine(MyMethod(5, 3));
    }

    // Outputs 8 (5 + 3)

}
```

You can also store the result in a variable (recommended, as it is easier to read and maintain):



```
using System;

class Program{

    static int MyMethod(int x, int y)
    {
      return x + y;
    }

    static void Main(string[] args)
    {
      int z = MyMethod(5, 3);
      Console.WriteLine(z);
    }

    // Outputs 8 (5 + 3)

}
```

C# - What is OOP

OOP stands for Object-Oriented Programming.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods).

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the C# code DRY **"Don't Repeat Yourself"**, and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

Tip: The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

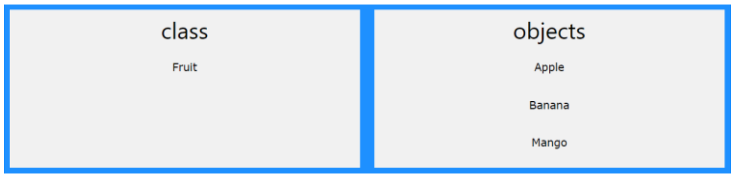
Encapsulation

Encapsulation binds together code and the data it manipulates and keeps them both safe from outside interference and misuse. Encapsulation is a protective container that prevents code and data from being accessed by other code defined outside the container.

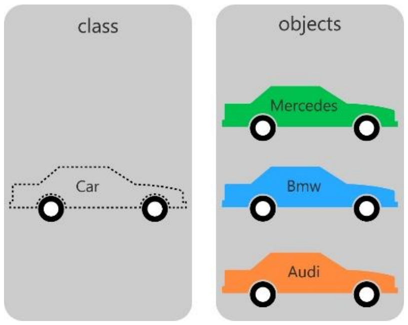
C# - What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:



Another Example:



So, a class is a template for objects, and an object is an instance of a class. When the individual objects are created, they inherit all the variables and methods from the class.

Create a Class

To create a class, use the class keyword:

Create a class named "*Car*" with a variable *color*:

```
class Car
{
  string color = "red";
}
```

When a variable is declared directly in a class, it is often referred to as a field (or attribute).

It is not required, but it is a good practice to start with an uppercase first letter when naming classes. Also, it is common that the name of the C# file and the class matches, as it makes our code organized. However, it is not required (like in Java).

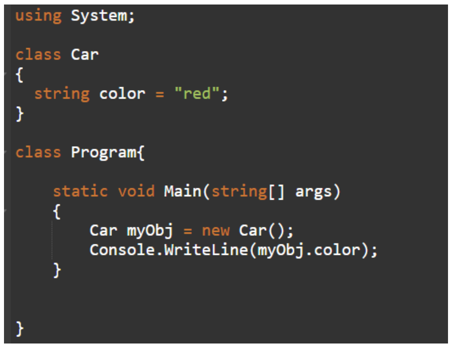
Create an Object

An object is created from a class. We have already created the class named Car, so now we can use this to create objects.

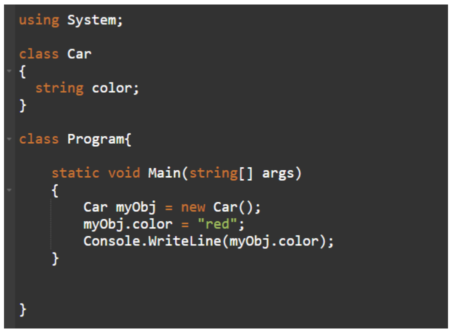
To create an object of Car, specify the class name, followed by the object name, and use the keyword new:

Example

Create an object called "myObj" and use it to print the value of color:



```
using System;

class Car
{
  string color = "red";
}

class Program{

    static void Main(string[] args)
    {
        Car myObj = new Car();
        Console.WriteLine(myObj.color);
    }


}
```



```
using System;

class Car
{
  string color;
}

class Program{

    static void Main(string[] args)
    {
        Car myObj = new Car();
        myObj.color = "red";
        Console.WriteLine(myObj.color);
    }


}
```

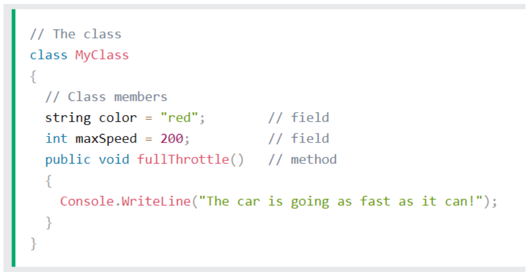
Note that we use the dot syntax (.) to access variables/fields inside a class (myObj.color).

Class Members

Fields and methods inside classes are often referred to as "Class Members":

Example

Create a Car class with three class members: two fields and one method.

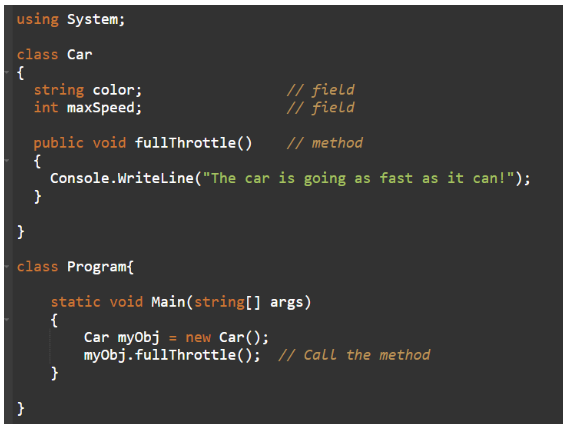


```
// The class
class MyClass
{
  // Class members
  string color = "red";        // field
  int maxSpeed = 200;          // field
  public void fullThrottle()   // method
  {
    Console.WriteLine("The car is going as fast as it can!");
  }
}
```

Object Methods

Methods normally belongs to a class, and they define how an object of a class behaves.

Just like with fields, you can access methods with the dot syntax. However, note that the method must be *public*. And remember that we use the name of the method followed by two parantheses *()* and a semicolon *;* to call (execute) the method:



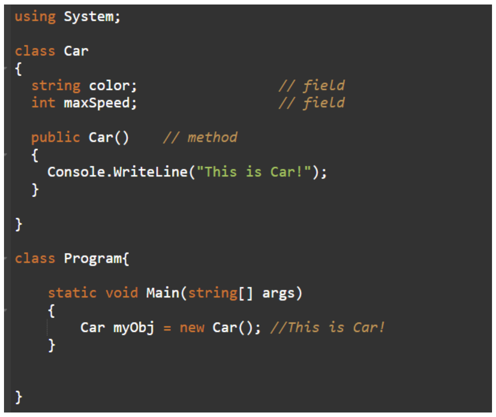
```
using System;

class Car
{
  string color;                 // field
  int maxSpeed;                 // field

  public void fullThrottle()    // method
  {
    Console.WriteLine("The car is going as fast as it can!");
  }

}

class Program{

    static void Main(string[] args)
    {
        Car myObj = new Car();
        myObj.fullThrottle();  // Call the method
    }

}
```

C# Constructors

A constructor is a **special method** that is used to initialize objects. The advantage of a constructor, is that it is called when an object of a class is created. It can be used to set initial values for fields:

Example

Create a constructor:



```
using System;

class Car
{
  string color;                 // field
  int maxSpeed;                 // field

  public Car()    // method
  {
    Console.WriteLine("This is Car!");
  }

}

class Program{

    static void Main(string[] args)
    {
        Car myObj = new Car(); //This is Car!
    }


}
```

Note that the constructor name must match the class name, and it cannot have a return type (like void or int).

Also note that the constructor is called when the object is created.

Access Modifiers

Example

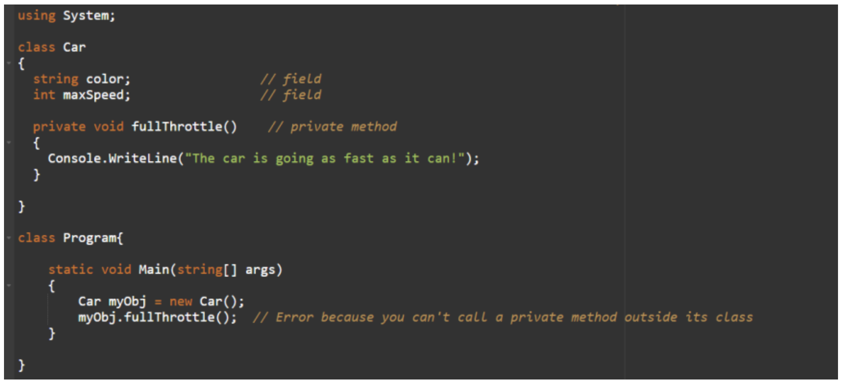
```
public string color;
```

The public keyword is an **access modifier**, which is used to set the access level/visibility for classes, fields, methods and properties.

C# has the following access modifiers:

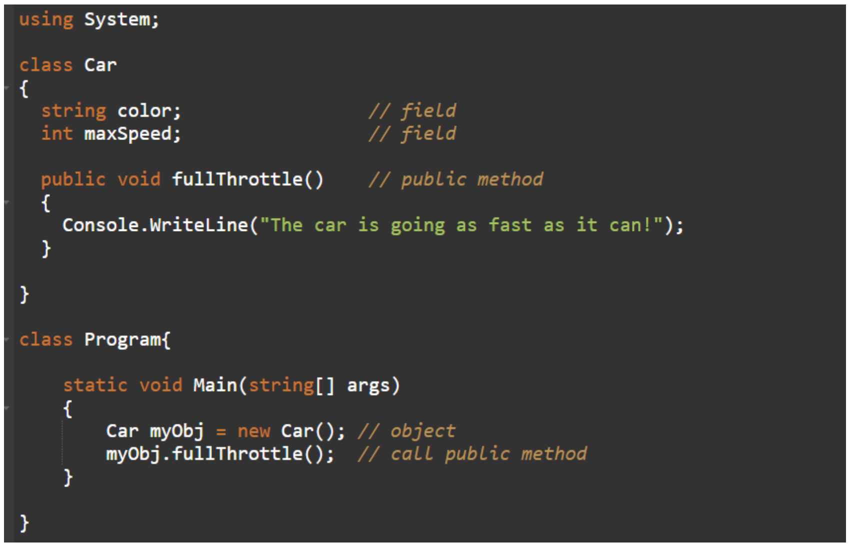
Modifier	Description
`public`	The code is accessible for all classes
`private`	The code is only accessible within the same class
`protected`	The code is accessible within the same class, or in a class that is inherited from that class. You will learn more about inheritance in a later chapter

Example:

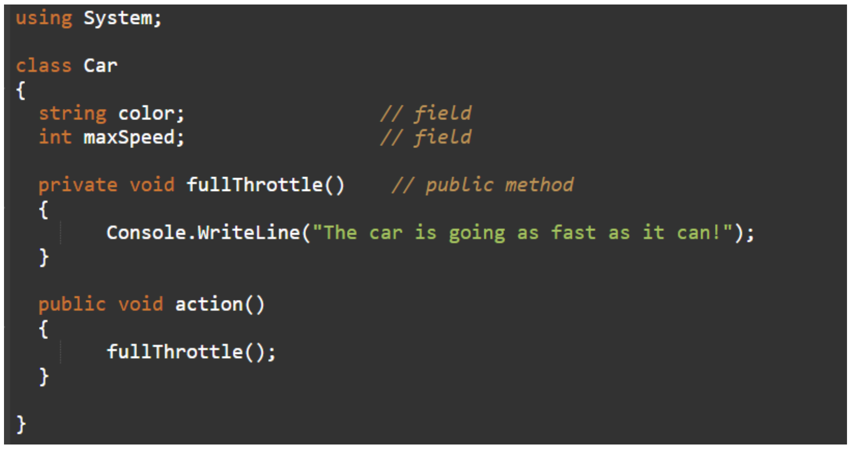


```
using System;

class Car
{
  string color;                 // field
  int maxSpeed;                 // field

  private void fullThrottle()    // private method
  {
    Console.WriteLine("The car is going as fast as it can!");
  }

}

class Program{

    static void Main(string[] args)
    {
        Car myObj = new Car();
        myObj.fullThrottle();  // Error because you can't call a private method outside its class
    }

}
```

Correct Example:



```
using System;

class Car
{
  string color;                 // field
  int maxSpeed;                 // field

  public void fullThrottle()    // public method
  {
    Console.WriteLine("The car is going as fast as it can!");
  }

}

class Program{

    static void Main(string[] args)
    {
        Car myObj = new Car(); // object
        myObj.fullThrottle();  // call public method
    }

}
```



```
using System;

class Car
{
  string color;                 // field
  int maxSpeed;                 // field

  private void fullThrottle()    // public method
  {
        Console.WriteLine("The car is going as fast as it can!");
  }

  public void action()
  {
        fullThrottle();
  }

}
```

Inheritance

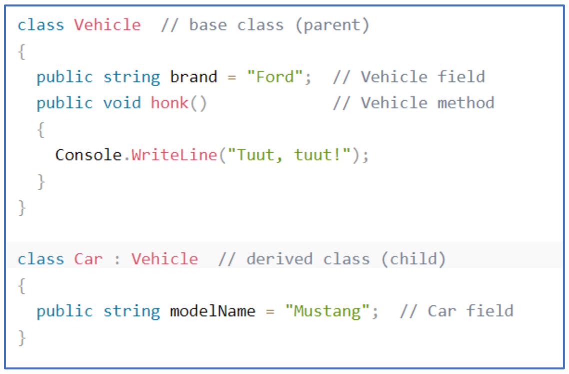
Inheritance (Derived and Base Class)

In C#, it is possible to inherit fields and methods from one class to another. We group the "inheritance concept" into two categories:

- Derived Class (child) - the class that inherits from another class
- Base Class (parent) - the class being inherited from

To inherit from a class, use the *:* symbol.

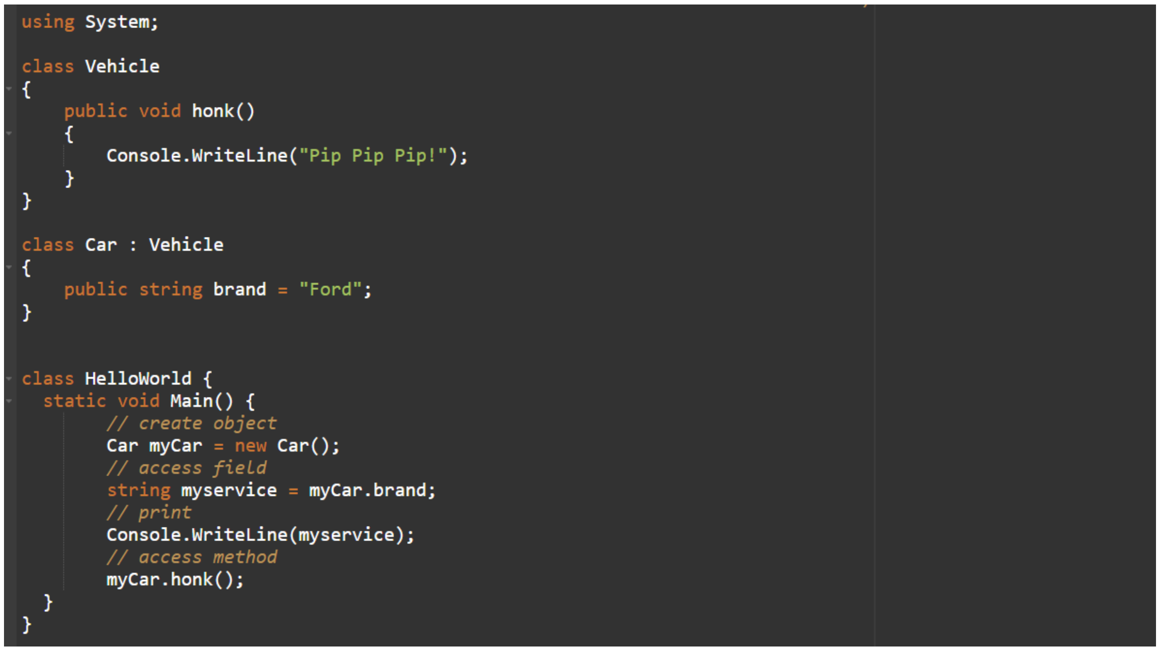
In the example below, the *Car* class (child) inherits the fields and methods from the *Vehicle* class (parent):



```
class Vehicle  // base class (parent)
{
  public string brand = "Ford";  // Vehicle field
  public void honk()             // Vehicle method
  {
    Console.WriteLine("Tuut, tuut!");
  }
}

class Car : Vehicle  // derived class (child)
{
  public string modelName = "Mustang";  // Car field
}
```

Why and When to Use "Inheritance"?

- It is useful for code reusability: reuse fields and methods of an existing class when you create a new class.



```
using System;

class Vehicle
{
    public void honk()
    {
        Console.WriteLine("Pip Pip Pip!");
    }
}

class Car : Vehicle
{
    public string brand = "Ford";
}


class HelloWorld {
  static void Main() {
        // create object
        Car myCar = new Car();
        // access field
        string myservice = myCar.brand;
        // print
        Console.WriteLine(myservice);
        // access method
        myCar.honk();
  }
}
```

As you can see in the example above, the **honk()** method is declared in **Vehicle** class but it can call using the **myCar object** of a **class Car**. This is because the properties and methods of the **Vehicle class** in *inherit* by the **Car class**.

Polymorphism

Polymorphism is a feature that allows one interface to be used for a general class of action. This concept is often expressed as "one interface, multiple actions". The specific action is determined by the exact nature of circumstances.

Polymorphism is the ability to treat the various objects in the same manner. It is one of the significant benefits of inheritance. We can decide the correct call at runtime based on the derived type of the base reference. This is called late binding.

Reusability

Once a class has been written, created and debugged, it can be distributed to other programmers for use in their own program. This is called reusability, or in .NET terminology this concept is called a component or a DLL. In OOP, however, inheritance provides an important extension to the idea of reusability. A programmer can use an existing class and without modifying it, add additional features to it.

References

https://www.w3schools.com/cs/cs_methods.php https://www.tutorialspoint.com/csharp/csharp_methods.htm

https://www.tutorialspoint.com/What-is-object-oriented-programming-OOP

https://www.educba.com/what-is-oop/

https://www.educative.io/blog/object-oriented-programming

https://searchapparchitecture.techtarget.com/definition/object-oriented-programming-OOP

https://www.computerhope.com/jargon/o/oop.htm

https://codewithgeeks.com/what-is-oop/

https://www.exercisescsharp.com/oop/

https://stackify.com/oop-concepts-c-sharp/

Object Oriented Programming Using C# .NET – Vidya Vrat Agarwal

https://www.codingame.com/playgrounds/52999/c-refresh/object-oriented-programming