C# OOP

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 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 easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

---

C# - What are Classes and Objects?

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

Class

Fruit

objects

Apple

Banana

Mango

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.

Classes and Objects

Everything in C# is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

A Class is like an object constructor, or a "blueprint" for creating objects.

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;

namespace MyApplication
{
  class Car
  {
    string color = "red";

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

```
red
```

Note that we use the dot syntax (`.`) to access variables/fields inside a class (`myObj.color`). You will learn more about fields in the next chapter.

Multiple Objects

You can create multiple objects of one class:

Example

Create two objects of `Car`:

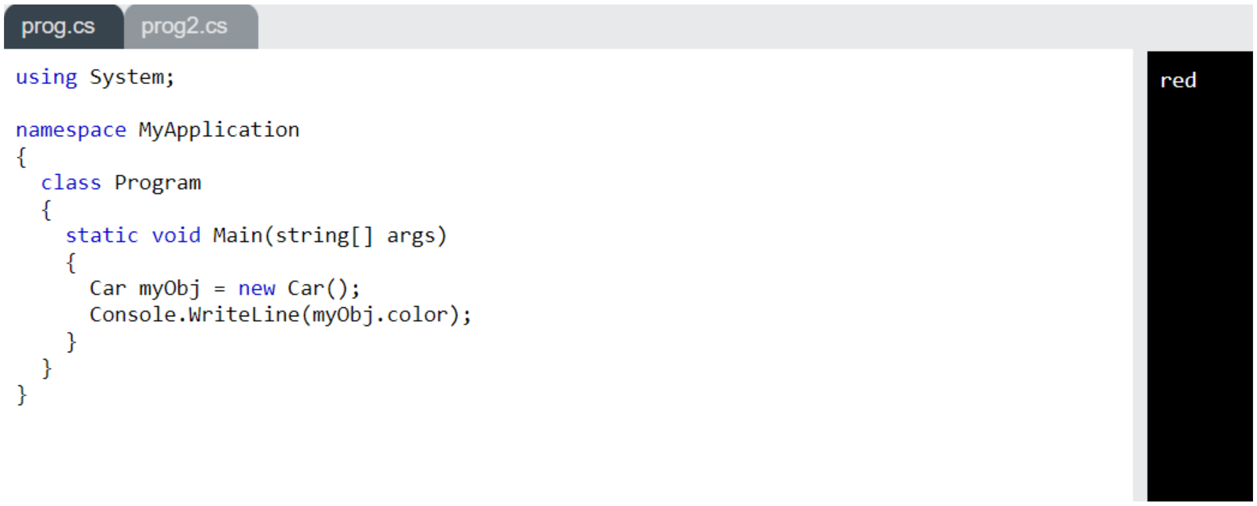
```
using System;

namespace MyApplication
{
  class Car
  {
    string color = "red";

    static void Main(string[] args)
    {
      Car myObj1 = new Car();
      Car myObj2 = new Car();
      Console.WriteLine(myObj1.color);
      Console.WriteLine(myObj2.color);
    }
  }
}
```

```
red
red
```

Using Multiple Classes

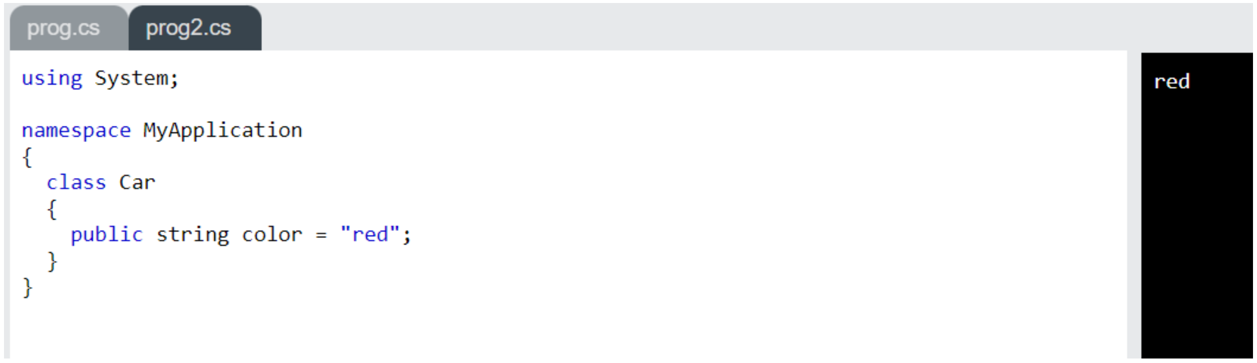
You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the fields and methods, while the other class holds the `Main()` method (code to be executed)).

- prog2.cs
- prog.cs



```
using System;

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

```
red
```



```
using System;

namespace MyApplication
{
  class Car
  {
    public string color = "red";
  }
}
```

```
red
```

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 drive()    // method

  {

    Console.WriteLine("The car is going as fast as it can!");

  }

}
```

Fields

We know that variables inside a class are called fields, and that you can access them by creating an object of the class, and by using the dot syntax (`.`).

The following example will create an object of the `Car` class, with the name `myObj`. Then we print the value of the fields `color` and `maxSpeed`:

Example

```
using System;

namespace MyApplication
{
  class Car
  {
    string color = "red";
    int maxSpeed = 200;

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

```
red
200
```

You can also leave the fields blank, and modify them when creating the object:

Example

```
//filename: Car.cs
using System;

namespace MyApplication
{
  class Car
  {
    string color;
    int maxSpeed;

    static void Main(string[] args)
    {
      Car myObj = new Car();
      myObj.color = "red";
      myObj.maxSpeed = 200;
      Console.WriteLine(myObj.color);
      Console.WriteLine(myObj.maxSpeed);
    }
  }
}
```

```
red
200
```

This is especially useful when creating multiple objects of one class:

Example

```
//filename: Car.cs
using System;

namespace MyApplication
{
  class Car
  {
    string model;
    string color;
    int year;

    static void Main(string[] args)
    {
      Car Ford = new Car();
      Ford.model = "Mustang";
      Ford.color = "red";
      Ford.year = 1969;

      Car Opel = new Car();
      Opel.model = "Astra";
      Opel.color = "white";
      Opel.year = 2005;

      Console.WriteLine(Ford.model);
      Console.WriteLine(Opel.model);
    }
  }
}
```

```
Mustang
Astra
```

Object Methods

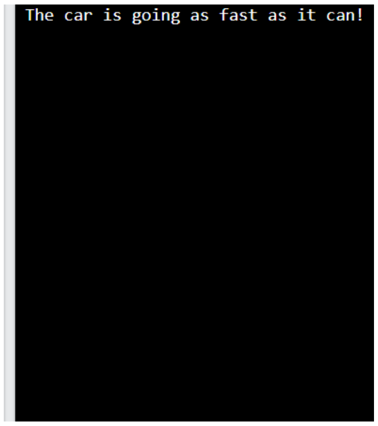
Methods are used to perform certain actions.

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:

Example

```
using System;

namespace MyApplication
{
  class Car
  {
    string color;                 // field
    int maxSpeed;                 // field
    public void fullThrottle()    // method
    {
      Console.WriteLine("The car is going as fast as it can!");
    }

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



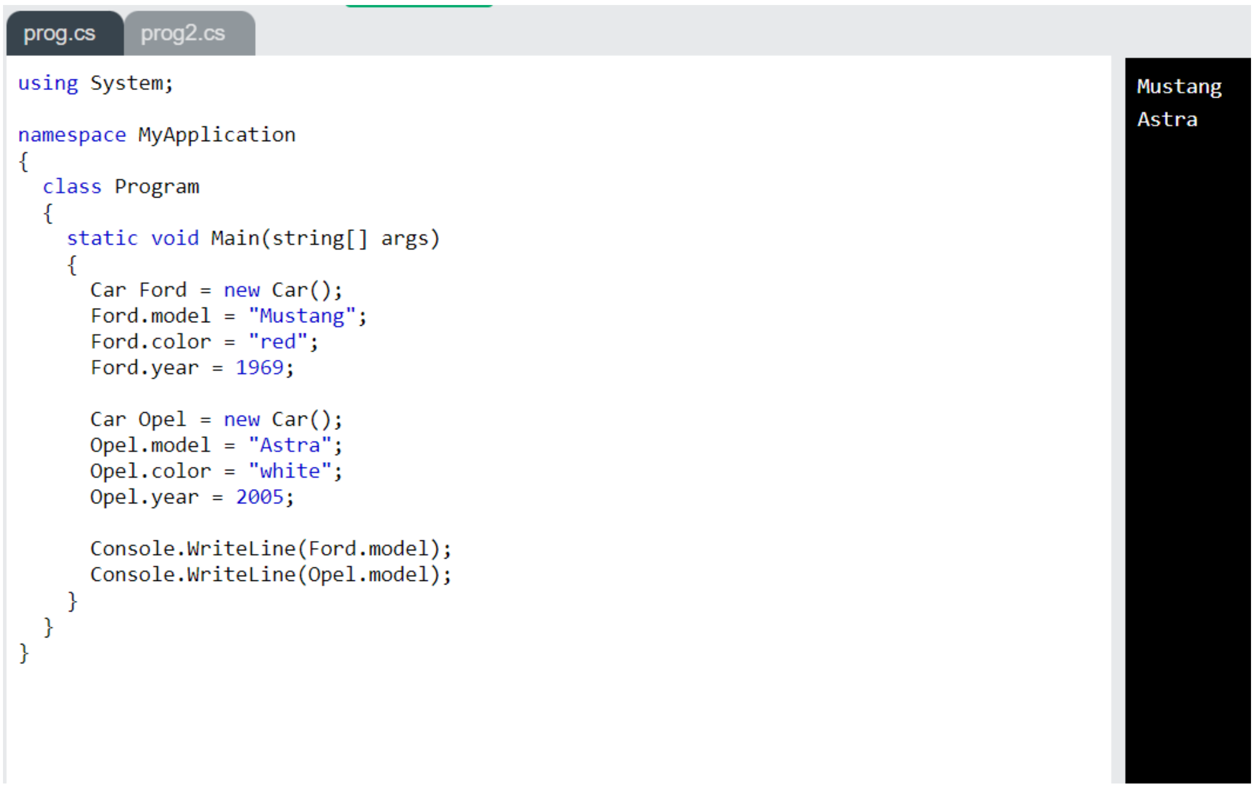
```
The car is going as fast as it can!
```

Why did we declare the method as `public`, and not `static`?

The reason is simple: a `static` method can be accessed without creating an object of the class, while `public` methods can only be accessed by objects.

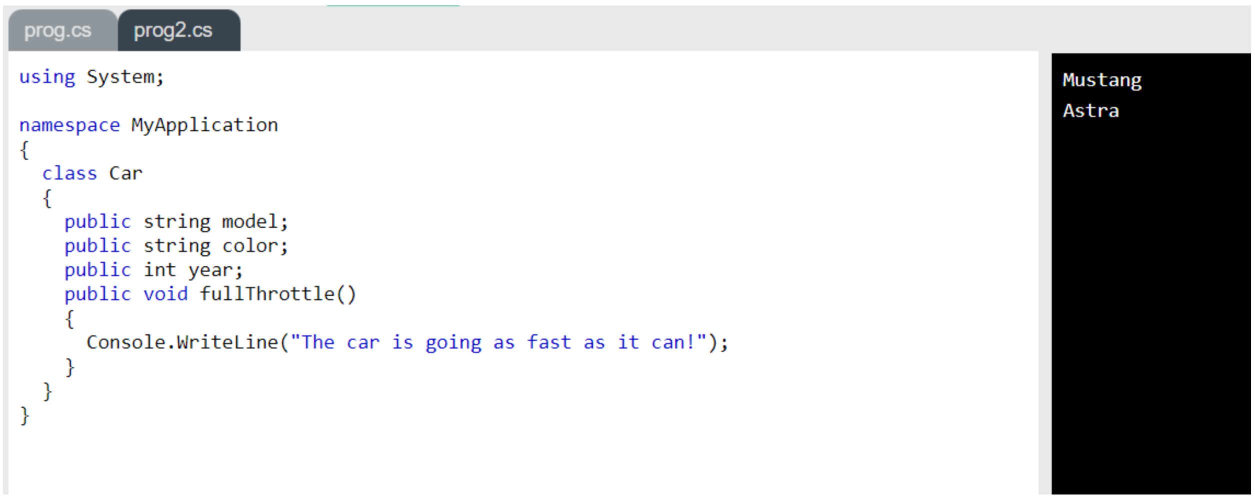
Use Multiple Classes

We can use multiple classes for better organization (one for fields and methods, and another one for execution). This is recommended:



```csharp
using System;

namespace MyApplication
{
  class Program
  {
    static void Main(string[] args)
    {
      Car Ford = new Car();
      Ford.model = "Mustang";
      Ford.color = "red";
      Ford.year = 1969;

      Car Opel = new Car();
      Opel.model = "Astra";
      Opel.color = "white";
      Opel.year = 2005;

      Console.WriteLine(Ford.model);
      Console.WriteLine(Opel.model);
    }
  }
}
```

```
Mustang
Astra
```



```csharp
using System;

namespace MyApplication
{
  class Car
  {
    public string model;
    public string color;
    public int year;
    public void fullThrottle()
    {
      Console.WriteLine("The car is going as fast as it can!");
    }
  }
}
```

```
Mustang
Astra
```

The `public` keyword is called an **access modifier**, which specifies that the fields of `Car` are accessible for other classes as well, such as `Program`.

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;

namespace MyApplication
{
  // Create a Car class
  class Car
  {
    public string model;  // Create a field

    // Create a class constructor for the Car class
    public Car()
    {
      model = "Mustang"; // Set the initial value for model
    }

    static void Main(string[] args)
    {
      Car Ford = new Car();  // Create an object of the Car Class (this will call the
constructor)
      Console.WriteLine(Ford.model);  // Print the value of model
    }
  }
}
```

```
Mustang
```

- 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.
- All classes have constructors by default: if you do not create a class constructor yourself, C# creates one for you. However, then you are not able to set initial values for fields.
- **Constructors save time**

Constructor Parameters

Constructors can also take parameters, which is used to initialize fields.

The following example adds a `string modelName` parameter to the constructor. Inside the constructor we set `model` to `modelName` (`model=modelName`). When we call the constructor, we pass a parameter to the constructor (`"Mustang"`), which will set the value of `model` to `"Mustang"`:

Example

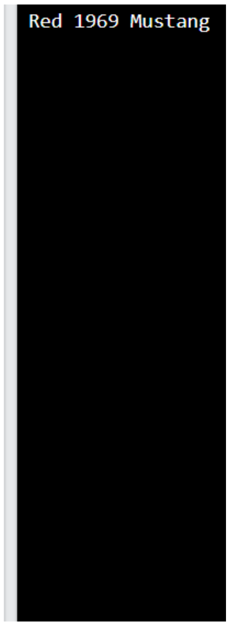
```
//filename: Car.cs
using System;

namespace MyApplication
{
  class Car
  {
    public string model;

    // Create a class constructor with a parameter
    public Car(string modelName)
    {
      model = modelName;
    }

    static void Main(string[] args)
    {
      Car Ford = new Car("Mustang");
      Console.WriteLine(Ford.model);
    }
  }
}
```

```
Mustang
```

You can have as many parameters as you want:

Example

```
using System;

namespace MyApplication
{
  class Car
  {
    public string model;
    public string color;
    public int year;

    // Create a class constructor with multiple parameters
    public Car(string modelName, string modelColor, int modelYear)
    {
      model = modelName;
      color = modelColor;
      year = modelYear;
    }

    static void Main(string[] args)
    {
      Car Ford = new Car("Mustang", "Red", 1969);
      Console.WriteLine(Ford.color + " " + Ford.year + " " + Ford.model);
    }
  }
}
```



```
Red 1969 Mustang
```

Just like other methods, constructors can be **overloaded** by using different numbers of parameters.

Constructors Save Time

When you consider the example from the previous chapter, you will notice that constructors are very useful, as they help reducing the amount of code:

Without constructor:

prog.cs

```
class Program

{

  static void Main(string[] args)

  {

    Car Ford = new Car();

    Ford.model = "Mustang";

    Ford.color = "red";

    Ford.year = 1969;
    Car Opel = new Car();
    Opel.model = "Astra";
    Opel.color = "white";
    Opel.year = 2005;

    Console.WriteLine(Ford.model);
    Console.WriteLine(Opel.model);
  }
}
```

With constructor:

prog.cs

```
class Program
{
  static void Main(string[] args)
  {
    Car Ford = new Car("Mustang", "Red", 1969);
    Car Opel = new Car("Astra", "White", 2005);

    Console.WriteLine(Ford.model);
    Console.WriteLine(Opel.model);
  }
}
```

C# Properties (Get and Set)

Properties and Encapsulation

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

- declare fields/variables as `private`
- provide `public` `get` and `set` methods, through **properties**, to access and update the value of a `private` field

---

Properties

`private` variables can only be accessed within the same class (an outside class has no access to it). However, sometimes we need to access them - and it can be done with properties.

A property is like a combination of a variable and a method, and it has two methods: a `get` and a `set` method:

Example

```
class Person

{

  private string name; // field

  public string Name   // property

  {

    get { return name; }   // get method

    set { name = value; }  // set method

  }

}
```

Example explained

- The Name property is associated with the name field. It is a good practice to use the same name for both the property and the private field, but with an uppercase first letter.

- The get method returns the value of the variable name.

- The set method assigns a value to the name variable. The value keyword represents the value we assign to the property.

- Now we can use the Name property to access and update the private field of the Person class:

Example

```
class Person

{

  private string name; // field

  public string Name   // property

  {

    get { return name; }

    set { name = value; }

  }

}


class Program

{

  static void Main(string[] args)

  {

    Person myObj = new Person();

    myObj.Name = "Liam";

    Console.WriteLine(myObj.Name);

  }

}
```

The output will be:

```
Liam
```

Inheritance (Derived and Base Class)

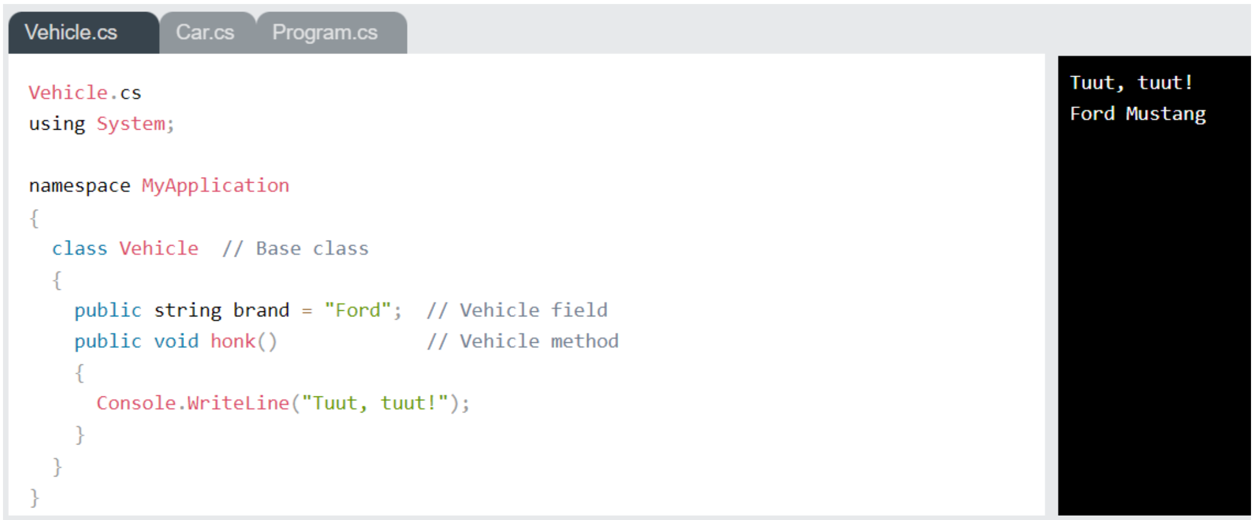
In C#, it is possible to inherit fields and methods from one class to another. We group the "inheritance concept" into two categories:

- **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.

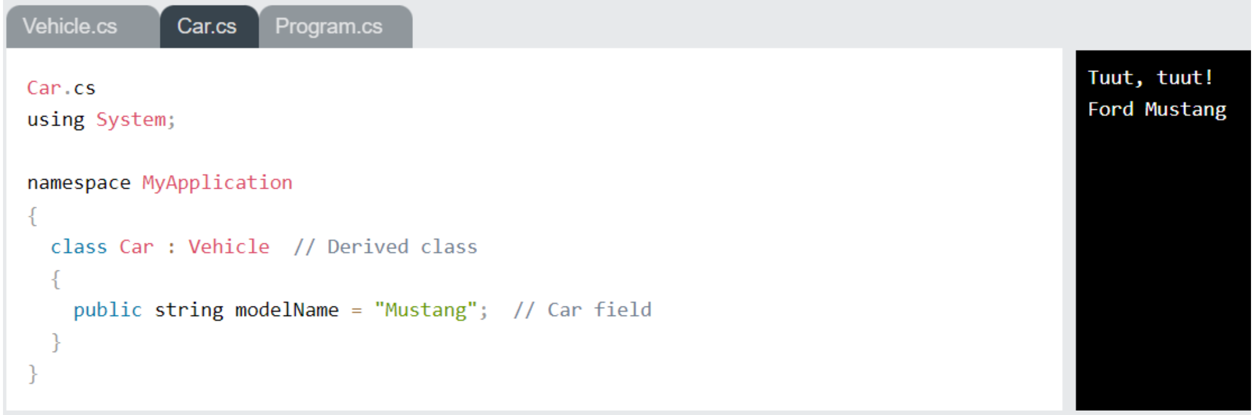
In the example below, the `Car` class (child) inherits the fields and methods from the `Vehicle` class (parent):

Example



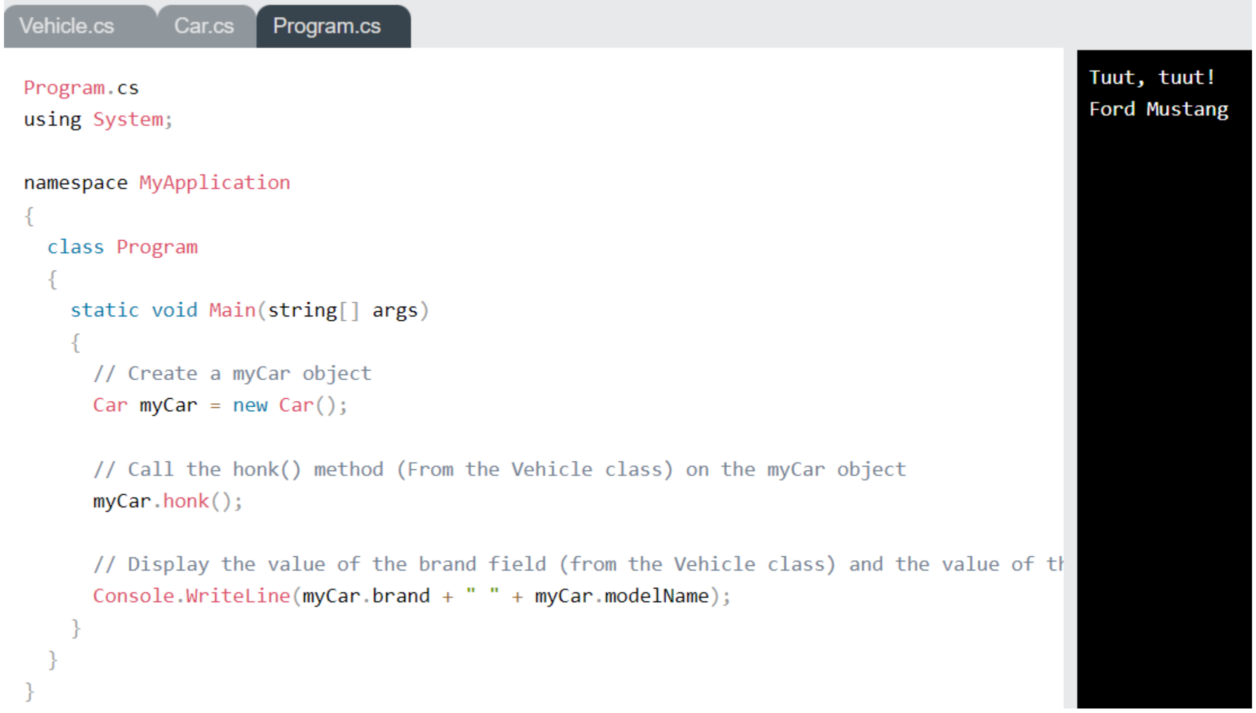
```
Vehicle.cs
using System;

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

```
Tuut, tuut!
Ford Mustang
```



```
Car.cs
using System;

namespace MyApplication
{
  class Car : Vehicle  // Derived class
  {
    public string modelName = "Mustang";  // Car field
  }
}
```

```
Tuut, tuut!
Ford Mustang
```



```
Program.cs
using System;

namespace MyApplication
{
  class Program
  {
    static void Main(string[] args)
    {
      // Create a myCar object
      Car myCar = new Car();

      // Call the honk() method (From the Vehicle class) on the myCar object
      myCar.honk();

      // Display the value of the brand field (from the Vehicle class) and the value of th
      Console.WriteLine(myCar.brand + " " + myCar.modelName);
    }
  }
}
```

```
Tuut, tuut!
Ford Mustang
```

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.

The sealed Keyword

If you don't want other classes to inherit from a class, use the `sealed` keyword:

If you try to access a `sealed` class, C# will generate an error:

```
sealed class Vehicle

{

  ...

}



class Car : Vehicle

{

  ...

}
```

The error message will be something like this:

```
'Car': cannot derive from sealed type 'Vehicle'
```

Polymorphism and Overriding Methods

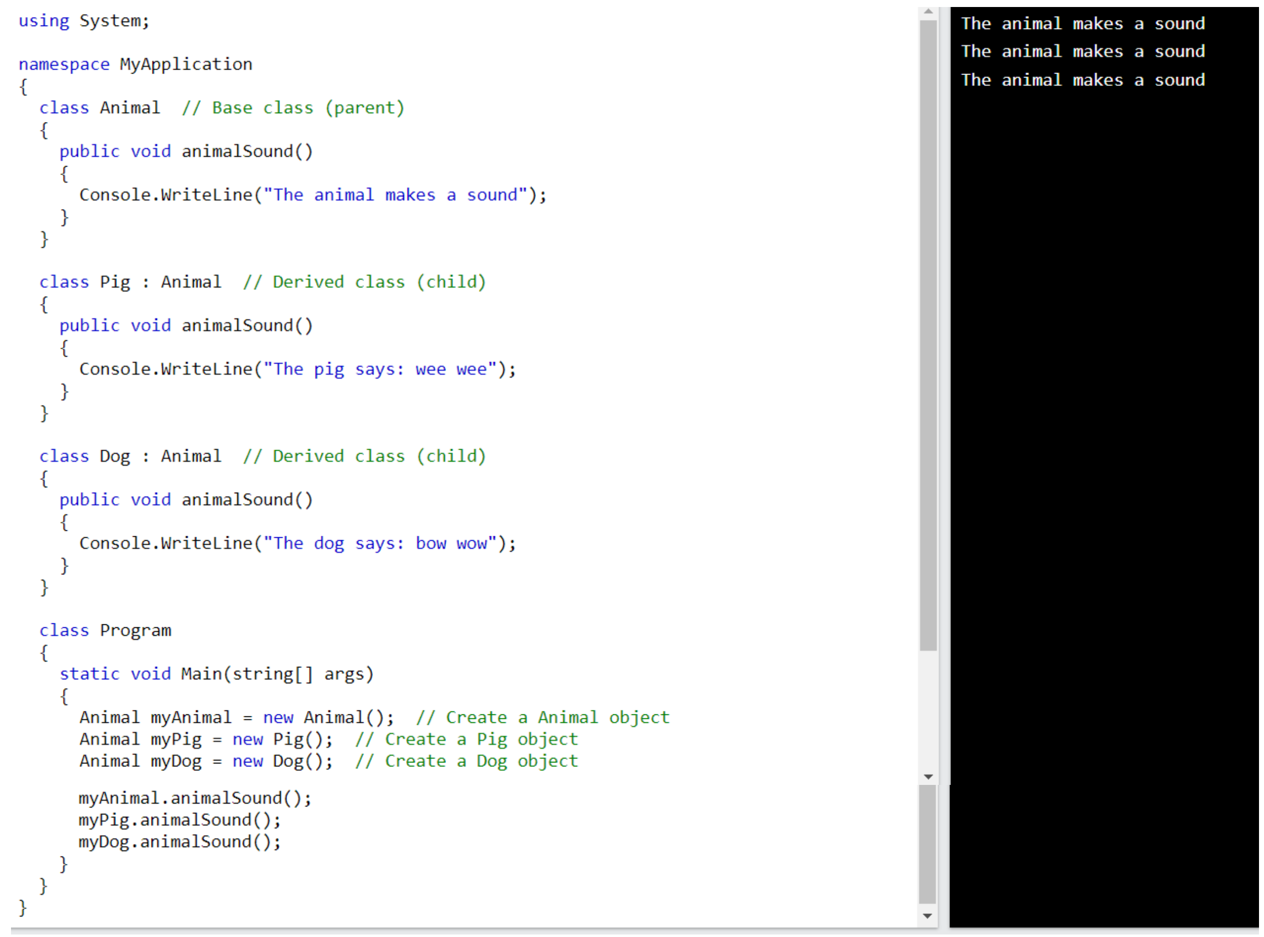
- Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.
- Like we specified in the previous chapter; **Inheritance** lets us inherit fields and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.
- For example, think of a base class called `Animal` that has a method called `animalSound()`. Derived classes of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

Example

```
class Animal  // Base class (parent)

{

  public void animalSound()

  {

    Console.WriteLine("The animal makes a sound");

  }

}


class Pig : Animal  // Derived class (child)

{

  public void animalSound()

  {

    Console.WriteLine("The pig says: wee wee");

  }

}


class Dog : Animal  // Derived class (child)

{
  public void animalSound()
  {
    Console.WriteLine("The dog says: bow wow");
  }
}
```

Now we can create `Pig` and `Dog` objects and call the `animalSound()` method on both of them:

Example



```
using System;

namespace MyApplication
{
  class Animal  // Base class (parent)
  {
    public void animalSound()
    {
      Console.WriteLine("The animal makes a sound");
    }
  }

  class Pig : Animal  // Derived class (child)
  {
    public void animalSound()
    {
      Console.WriteLine("The pig says: wee wee");
    }
  }

  class Dog : Animal  // Derived class (child)
  {
    public void animalSound()
    {
      Console.WriteLine("The dog says: bow wow");
    }
  }

  class Program
  {
    static void Main(string[] args)
    {
      Animal myAnimal = new Animal();  // Create a Animal object
      Animal myPig = new Pig();  // Create a Pig object
      Animal myDog = new Dog();  // Create a Dog object

      myAnimal.animalSound();
      myPig.animalSound();
      myDog.animalSound();
    }
  }
}
```

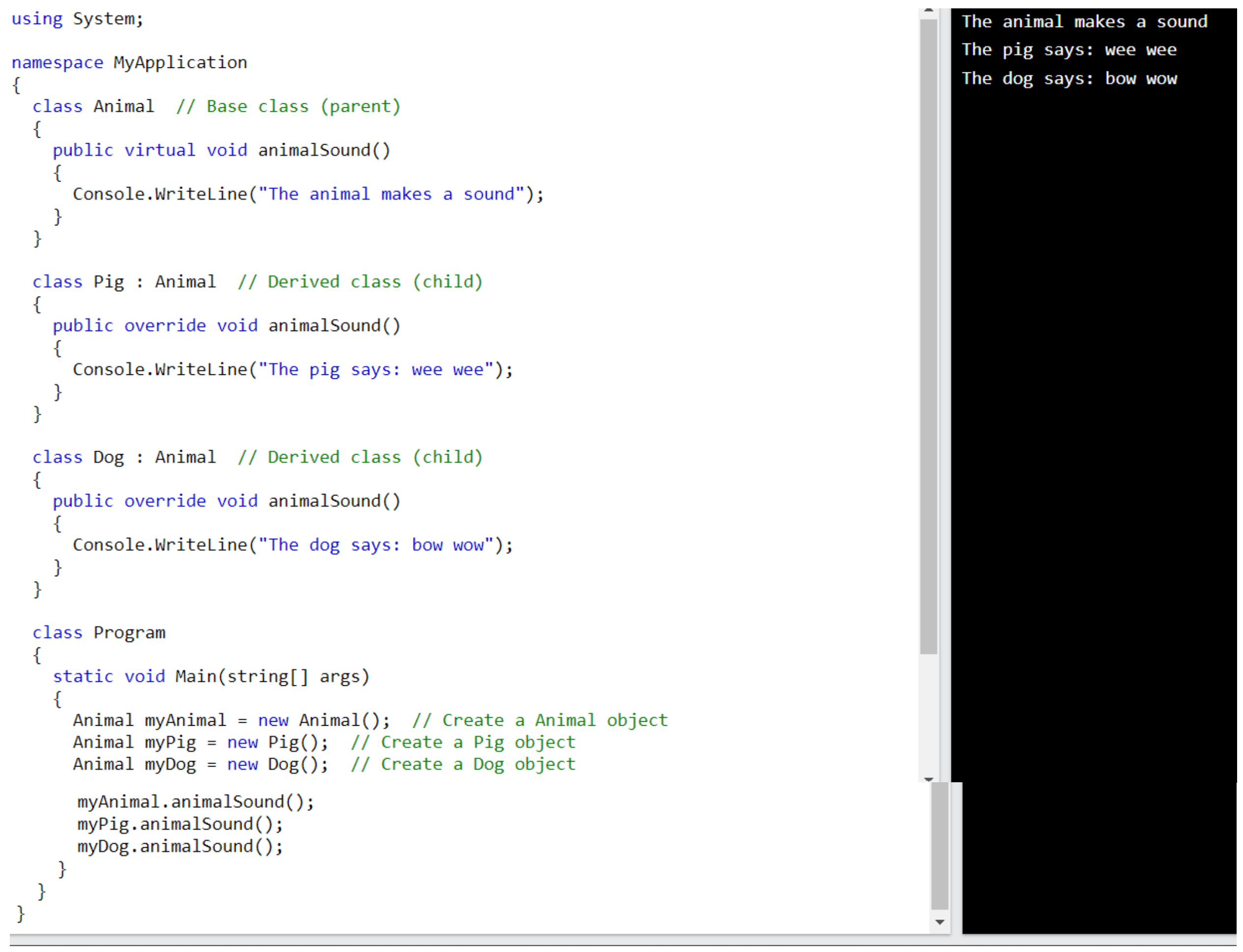
```
The animal makes a sound
The animal makes a sound
The animal makes a sound
```

Not The Output I Was Looking For

The output from the example above was probably not what you expected. That is because the base class method overrides the derived class method, when they share the same name.

However, C# provides an option to override the base class method, by adding the `virtual` keyword to the method inside the base class, and by using the `override` keyword for each derived class methods:

Example



```
using System;

namespace MyApplication
{
  class Animal  // Base class (parent)
  {
    public virtual void animalSound()
    {
      Console.WriteLine("The animal makes a sound");
    }
  }

  class Pig : Animal  // Derived class (child)
  {
    public override void animalSound()
    {
      Console.WriteLine("The pig says: wee wee");
    }
  }

  class Dog : Animal  // Derived class (child)
  {
    public override void animalSound()
    {
      Console.WriteLine("The dog says: bow wow");
    }
  }

  class Program
  {
    static void Main(string[] args)
    {
      Animal myAnimal = new Animal();  // Create a Animal object
      Animal myPig = new Pig();  // Create a Pig object
      Animal myDog = new Dog();  // Create a Dog object

      myAnimal.animalSound();
      myPig.animalSound();
      myDog.animalSound();
    }
  }
}
```

```
The animal makes a sound
The pig says: wee wee
The dog says: bow wow
```

Why And When To Use "Inheritance" and "Polymorphism"?

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

C# Abstraction

Abstract Classes and Methods

- Data **abstraction** is the process of hiding certain details and showing only essential information to the user.
- Abstraction can be achieved with either **abstract classes** or **interfaces**.

The `abstract` keyword is used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the derived class (inherited from).

An abstract class can have both abstract and regular methods:

```
abstract class Animal

{

  public abstract void animalSound();

  public void sleep()

  {

    Console.WriteLine("Zzz");

  }

}
```

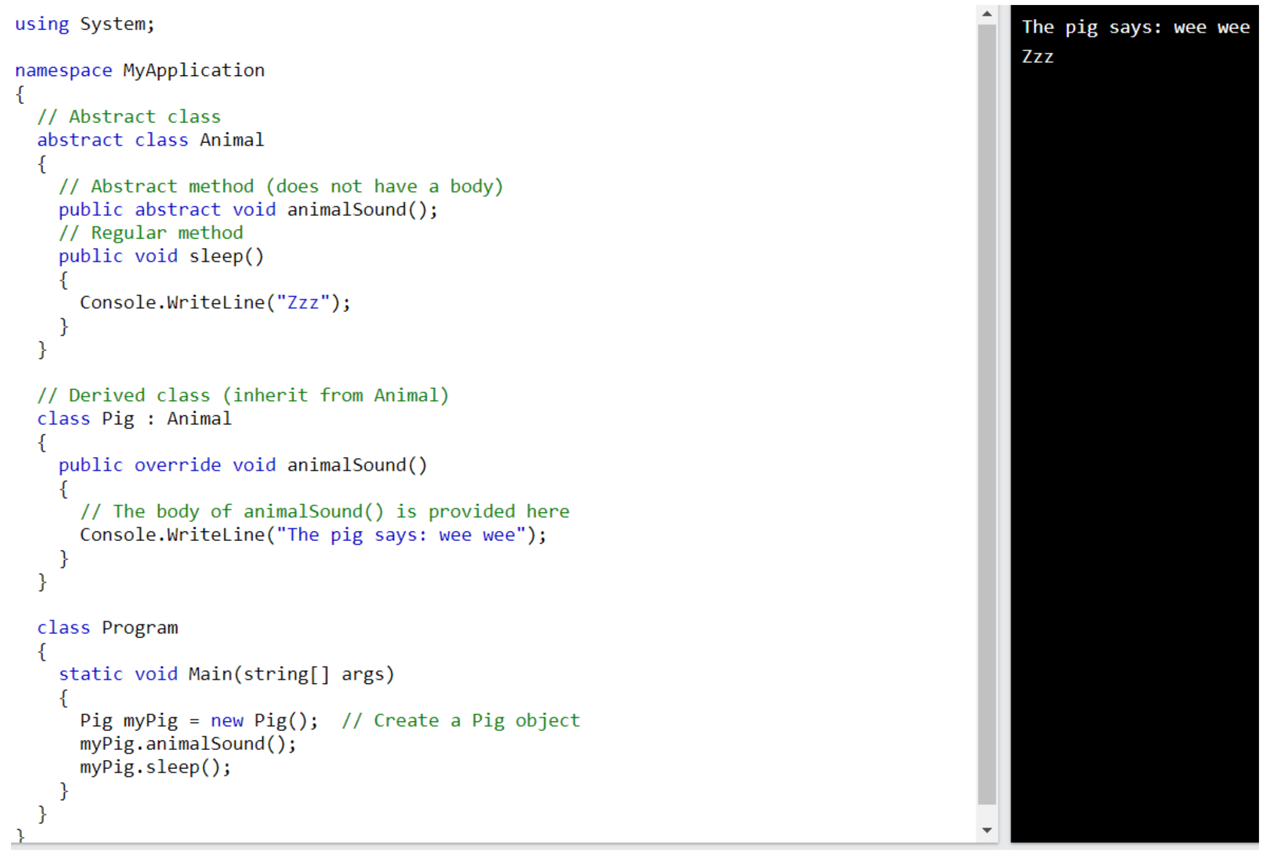
From the example above, it is not possible to create an object of the Animal class:

```
Animal myObj = new Animal(); // Will generate an error (Cannot create an instance of the
abstract class or interface 'Animal')
```

To access the abstract class, it must be inherited from another class.

we use the : symbol to inherit from a class, and that we use the override keyword to override the base class method.

Example



```
using System;

namespace MyApplication
{
  // Abstract class
  abstract class Animal
  {
    // Abstract method (does not have a body)
    public abstract void animalSound();
    // Regular method
    public void sleep()
    {
      Console.WriteLine("Zzz");
    }
  }

  // Derived class (inherit from Animal)
  class Pig : Animal
  {
    public override void animalSound()
    {
      // The body of animalSound() is provided here
      Console.WriteLine("The pig says: wee wee");
    }
  }

  class Program
  {
    static void Main(string[] args)
    {
      Pig myPig = new Pig();  // Create a Pig object
      myPig.animalSound();
      myPig.sleep();
    }
  }
}
```

```
The pig says: wee wee
Zzz
```

Why And When To Use Abstract Classes and Methods?

To achieve security - hide certain details and only show the important details of an object.

Note: Abstraction can also be achieved with Interfaces