# Object Oriented Programming Concepts

## Objects and classes

Object oriented programming (OOP) is a very important topic in .NET programming. In object oriented programming the object is the most important concept it is a self-contained entity that has state and behaviour.

In programming an object's state is described as its fields and properties, and its behaviour is defined by its methods and events. OOP helps programmers to design their applications.

We often use objects in programs to describe real-world objects we can have objects of type car, customer, document, or person. Each object has its own state and behaviour.

It is important to understand the difference between a class and an object. A class acts like a blueprint for the object, and an object represents an instance of the class. In the case of objects of type car for example the class or type is Car. We can create as many Car objects as we want and give them names such as myCar, JohnsCar etc.

The class also called the type defines the behaviours and attributes that will apply to all objects of that type. All objects of type car will have the same behaviour for example change gear. However each individual car may be in a different gear at any particular time.

### Creating an object

You can create an object the same way you create a variable.

```
string name;
int age;

Car myCar;
```

Activity 1. Creating a class

*Person1.cs & personTest.aspx*

Create a new website and add the App_code folder to it. Add a new class and name it Person.

The code should look like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

/// <summary>
/// Summary description for Person
/// </summary>
public class Person
{
	public Person()
	{
		//
		// TODO: Add constructor logic here
		//
	}
}
```

We will add 2 public fields:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

/// <summary>
/// Summary description for Person
/// </summary>
public class Person
{
    public int Age;
    public string Name;

	public Person()
```

```
        {
                //
                // TODO: Add constructor logic here
                //
        }
}
```

The 2 fields Age and Name are marked with the public access modifier which means they can be accessed from anywhere that uses objects of this class.

### Using the person class

In order to use the person class we need a program that will create Person objects.

Create the following file to use the person class:

Add a label named lblMessage

In the code behind add the following code:

```
protected void Page_Load(object sender, EventArgs e)
{
        Person Jane;
        Jane = new Person();


        Jane.Age = 20;
        Jane.Name = "Jane Potter";

        lblMessage.Text = Jane.Name + "<br />" + Jane.Age;
}
```

The code in detail:

| | |
|---|---|
| `Person Jane;` | Declare object of type person |
| `Jane = new Person();` | Because person is not a simple type you need to create a new instance of it. |
| `Jane.Age = 20;`<br>`Jane.Name = "Jane Potter";` | Assign values to the public fields |
| `lblMessage.Text = Jane.Name + "<br />" + Jane.Age;` | Read the field values and display them in a label |

## Adding behaviour to the person class

Objects encapsulate both state (instance data) and behaviour. In the person class Age and Name are instance data. Behaviour is implemented with methods.

### Activity 2. Adding a method

*Person2.cs & PersonTest2.aspx*

Add a method to the Person class

```
public class Person
{
    public int Age;
    public string Name;

       public Person()
       {
                //
                // TODO: Add constructor logic here
                //
       }

    public string DescribeYourself()
    {
        string strDescription;

        strDescription = "My name is " + Name + " and I am " + Age + " years old";

        return strDescription;
```

```
    }
}
```

To test the method modify your test file to use the following code:

```
protected void Page_Load(object sender, EventArgs e)
{
        Person Jane;
        Jane = new Person();


        Jane.Age = 20;
        Jane.Name = "Jane Potter";

        lblMessage.Text = Jane.DescribeYourself();
}
```

## Access modifiers

In the person class all the members (properties, methods and events) are marked with the public access modifier:

```
public int Age;
public string Name;


public string DescribeYourself()
…
```

This means that other classes that use the person class will have full access to these members by using the following syntax:

```
Jane.Age = 20;
Jane.Name = "Jane Potter";
Jane.DescribeYourself();
```

The syntax is:

objectName.PropertyName

objectName.MethodName()

Method have brackets properties don't.

The other access modifier we will discuss at this point is private. A private class member is only accessible from within the class it is defined. For all other classes it is invisible.

The classes public interface is defined by its public members.

It is recommended when designing your class to make all members private and only make public the ones that really need to be public.

To test out the difference between public and private change the Age variable to private and see how it affects the file which uses the person class.

There are other access modifiers:

Public – The least restrictive access modifier. Outer classes have full access to a public member

Private – A private member is accessible only to the code in the current class.

Protected – A protected member is accessible to the code within the class or to classes derived from it.

Internal – An internal member is accessible from the current assembly (current project)

Protected internal – A protected internal member is accessible from the current project or from classes derived from its containing class.

Protected, internal and protected internal are not relevant for us as we are creating websites.

## Adding properties

Properties provide a way for an object to control and validate the values saved to and retrieve from its fields.

At the moment our person class has public fields this means any value can be allocated to the age and name variables. Even incorrect values. With properties we can control how the fields are accessed.

### Activity 3. Adding properties to the person class

*Person3.cs & personTest3.aspx*

Update the person class to contain some properties:

```
public class Person
{
    private int _age;
    private string _name;

    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            value = value.Trim();
            _name = value;
        }
    }

    public int Age
    {
        get
        {
            return _age;
        }
        set
        {
            if (value < 0)
            {
                throw new Exception("Age must be a positive value");
            }
            else
            {
                _age = value;
            }
        }
    }
```

…

To test it try and put in a negative age value and then try and store “ Jane Potter “ in the name value.

Using properties is a very effective way to control what goes in or out of our object. It is often a good idea to use properties even for values that are directly stored in the private fields without any validation or calculation.

## Read only and write only properties

A read only property is a property without a set accessor. A write only property is a property without a get accessor.

A private value does not need to have a matching public property and a property does not need to match to a private value for example you may have a class called rectangle with an area property defined as:

```
Public int Area
{
       get
       {
              Return height * width;
       }
}
```

In the above code example set is missing so the area property is read only.

### Activity 4. Read and write only

*Person4.cs & personTest4.aspx*

Change the class so that the age and name are write only:

```
public string Name
    {
```

```
        set
        {
            value = value.Trim();
            _name = value;
        }
    }

    public int Age
    {

        set
        {
            if (value < 0)
            {
                throw new Exception("Age must be a positive value");
            }
            else
            {
                _age = value;
            }
        }
    }
```

To test it try and display the Age and Name properties to a label:

```
lblMessage.Text = Jane.Age;
```

You should get an error.

# Principles of object oriented programming

## Encapsulation

Encapsulation is the separation between an objects exposed functionality and its internal structure. The separation between what an object does and how the object does it. What an object does is represented by the public interface, the public members of the class. How an object does it is made up by the class's private members, methods and property implementations. We face this in everyday life, you know how to use a vending machine by interacting with the interface (pressing buttons etc) but you don't need to see or know how the internal working of the machine dispense the item and provide change. This concept is the same for programming. Other programmers need to know the available properties, methods and events but they don't need to know how the methods are implemented. For example when we need to bind data to a gridview we need to know there is a method called dataBind() which carries out that functionality but we don't need to know how it is done.

So far our Person class can be represented with the following class diagram:

| **Person** | |
|---|---|
| -_age: int<br>-_name: string | Class's fields |
| +Age(): int<br>+Name() : string<br>+DescribeYourSelf() | Methods and properties<br>Age() and Name() are called **accessor methods** (also **getting** and **setting** methods) and need not be shown in your model as they are usually inferred. |

Each member has a + or – on the left. A + means it is public, a – means it is private.

Age, Name and DescribeYourSelf can be used and called by outer classes such as webforms. _age and _name can only be used inside the Person class file.

One of the features of OOP is the ability to reuse classes. Classes are created as self-contained units. When you build a class for example a person class you can later decide to use it in a completely different application. Protecting the members by setting them as private ensures the class data is protected so that code cannot access it in a way it wasn't intended.

If we go back to the vending machine example we can easily pick up the machine and move it to a different location and it will still work exactly the same way. Now imagine the left side panel of the vending machine was not sealed making the cans publicly accessible. If the vending machine was placed in location A with its left side against a wall making the cans inaccessible when it is moved to location B we have no control over where it is placed possibly

making the left side accessible. This is the same concept as making members private, by making them private we are sure they will never be accessed directly and therefore protecting the internal workings of the class.

## Inheritance

Inheritance is a very important feature of OOP, it enables a class to inherit all the properties and behaviour of another class while permitting the class to modify the inherited behaviour or add its own. The class that inherits is called a derived class, while the class being inherited from is said to be the base class.

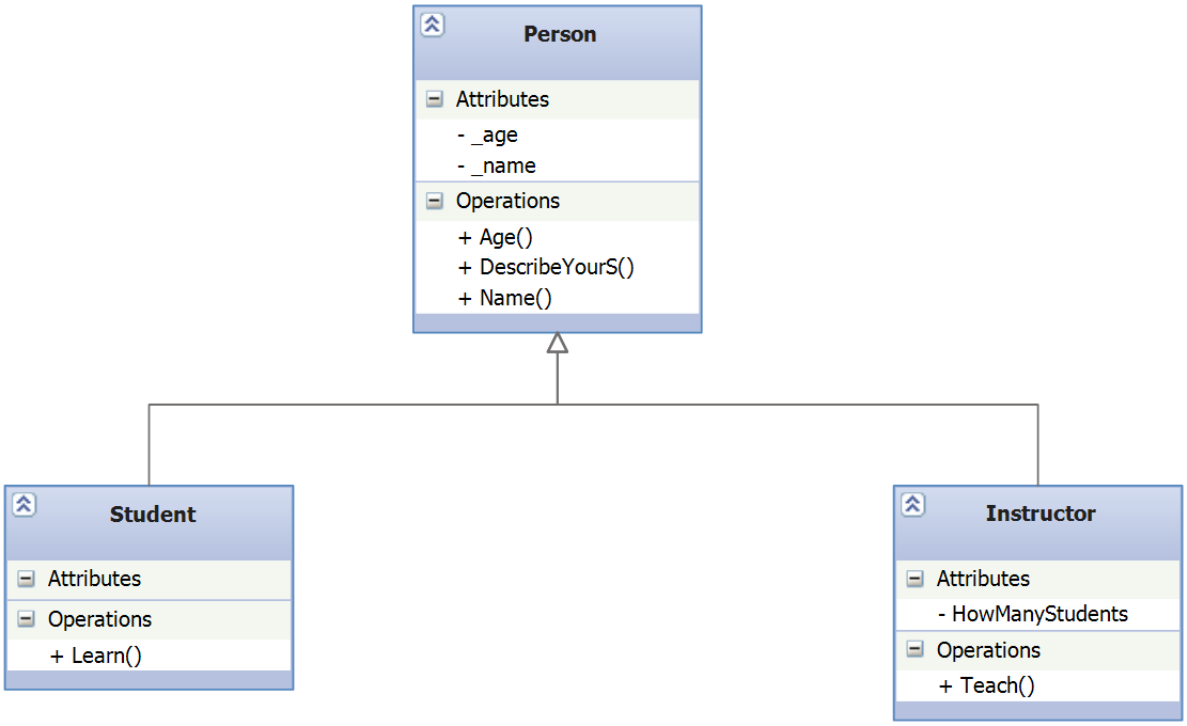
When you have a large number of classes it is possible that some of them may be related to others in some way. If we take an example where we need to implement 2 classes a student and an instructor. In the specifications student and instructor both have an age and name property and a describe yourself method. Student needs an additional method called learn and instructor needs an additional method called teach and a property called HowManyStudents.

You can create these 2 classes separately:

| Student |
|---|
| -_age: int<br>-_name: string |
| +Age()<br>+Name()<br>+DescribeYourself()<br>+Learn() |

| Instructor |
|---|
| -_age: int<br>-_name: string<br>-_HowManyStudents: int |
| +Age()<br>+Name()<br>+DescribeYourself()<br>+Teach() |

If we have a look at the properties and methods we can see that both classes have some common functionality. We can code these separately but there would be a lot of redundant code. Instead what we can do is create a base or parent class containing the common properties and methods and have the student and instructor class inherit from it.




As you can see the derived classes only implement the additional functionality.

Inheritance can be applied when a class is a kind of base class. For example a student is a kind of person so is an instructor.

## Inheritance vs composition

Inheritance is used to create a specialised version of a class that already exists. Student and instructor are special kinds of person. With inheritance all public members of the base class automatically become part of the derived class.

Composition refers to using one or more objects as fields of another class. The new class is said to be composed of other objects. Composition is verified with a has a rule. For example if we have a car class and an engine class we can say that a car has an engine.

### Activity 5. Inheritance

In this exercise we will create a student class and an instructor class. Both of these classes inherit from the Person class.

Student class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

/// <summary>
/// Summary description for Student
/// </summary>
public class Student : Person
{
        public Student()
        {
                //
                // TODO: Add constructor logic here
                //
        }

    public string Learn()
    {
        string strLearn;

        strLearn = "I am ready to learn";

        return strLearn;
    }
}
```

Instructor class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

/// <summary>
/// Summary description for Instructor
/// </summary>
public class Instructor : Person
{
        public Instructor()
        {
                //
                // TODO: Add constructor logic here
                //
        }

    public string Teach()
    {
        string strTeach;

        strTeach = "Let's begin";
```

```
        return strTeach;
    }
}
```

To test the page you will need to create a web form with 2 labels lblTeacher and lblStudent

```
protected void Page_Load(object sender, EventArgs e)
{
        Instructor theTeacher = new Instructor();

        theTeacher.Age = 30;
        theTeacher.Name = "Kate Smith";

        Student newStudent = new Student();
        newStudent.Age = 20;
        newStudent.Name = "Jackie Howard";

        lblTeacher.Text = theTeacher.DescribeYourself();
        lblTeacher.Text += "<br />" + theTeacher.Teach();
        lblStudent.Text = newStudent.DescribeYourself();
        lblStudent.Text += "<br />" + newStudent.Learn();


}
```

We can see that both the Instructor and Student can describe themselves in the same way that the Person class describes themselves.

When you use inheritance you can access all the objects members and any member of the parent or base class.

Inheritance can save you lines of code, it allows us to reuse some functionality.

## Polymorphism

Polymorphism means that one thing has the ability to take many shapes. In programming terms this means a method can have many behaviours. A simple example is an on/off switch, which could be on a light switch, computer, radio, printer etc. Regardless of where the switch is, it is used the same way, you switch it on to turn on the device/lights etc or off. The internal workings of switching a light on or off are very different to switching a computer on or off. So we have the same action name (on or off) and the same affect (switch on or switch off) but the internal workings are very different. Polymorphism is when you have two or more same named items (on/off) performing the same task (turn on/turn off) despite being very different internally.

### Activity 6. Polymorphism

To best understand this create a new file with 2 labels lblStudent and lblTeacher. Add the following code to the code behind:

```
protected void Page_Load(object sender, EventArgs e)
{
        Instructor theTeacher = new Instructor();

        theTeacher.Age = 30;
        theTeacher.Name = "Kate Smith";

        Student newStudent = new Student();
        newStudent.Age = 20;
        newStudent.Name = "Jackie Howard";

        lblTeacher.Text = DescribePerson(theTeacher);
        lblStudent.Text = DescribePerson(newStudent);
}

private string DescribePerson(Person objPerson)
{
        return objPerson.DescribeYourself();
}
```

The describe person method needs as input an object of type Person yet we are passing it an object of type Instructor and an object of type Student. Inside the DescribePerson method we can call any method of the person

class. Although the parameter is of type person we can pass it anything which is derived from the person class (anything that inherits from person).

Polymorphism is powerful in that we can define a generic method like DescribePerson without knowing what kind of person we will receive.

Once the Student or Instructor objects are passing into the method as Person we can no longer use the student or instructor methods only the person methods.

## Method overloading

Method overloading allows us to have many methods with the same name, but different types of parameters.

### Activity 7. Overloading

Add an extra method to the Person class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

/// <summary>
/// Summary description for Person
/// </summary>
public class Person
{
    private int _age;
    private string _name;

    public string Name
    {

        set
        {
            value = value.Trim();
            _name = value;
        }

        get
        {
            return _name;
        }
    }

    public int Age
    {

        set
        {
            if (value < 0)
            {
                throw new Exception("Age must be a positive value");
            }
            else
            {
                _age = value;
            }
        }

        get
        {
            return _age;
        }
    }

       public Person()
       {
              //
```

```
            // TODO: Add constructor logic here
            //
        }

    public string DescribeYourself()
    {
        string strDescription;

        strDescription = "My name is " + Name + " and I am " + Age + " years old";

        return strDescription;
    }

    public string DescribeYourself(string somebody)
    {
        string strDescription;

        strDescription = "Hello " + somebody + "My name is " + Name + " and I am " + Age + " years
old";

        return strDescription;
    }
}
```

As you can see we have 2 DescribeYourself methods. To test it create a page with a label called lblMessage and add the following code to the code behind:

```
protected void Page_Load(object sender, EventArgs e)
    {
        Person Jane;
        Jane = new Person();


        Jane.Age = 20;
        Jane.Name = "Jane Potter";

        lblMessage.Text = Jane.DescribeYourself();

        lblMessage.Text += "<br />" + Jane.DescribeYourself("John");
    }
```

We are making use of both DecribeYourself methods. The compiler will decide which one to use based on the parameter list. The parameters either have to have different data types or a different number of parameters.

## Constructors

Constructors are a special type of method used to initialise the object. The constructor is always the first method that gets called when the object is created using the new keyword. The only 2 ways to call a constructor are to create an object or call the constructor from another constructor.

The constructor is created as a method that has the same name as the class and has no return type, not even void.

### Activity 8. Constructors

Modify Person to contain a constructor.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

/// <summary>
/// Summary description for Person
/// </summary>
public class Person
{
    private int _age;
    private string _name;

    public string Name
```

```
    {

        set
        {
            value = value.Trim();
            _name = value;
        }

        get
        {
            return _name;
        }
    }

    public int Age
    {

        set
        {
            if (value < 0)
            {
                throw new Exception("Age must be a positive value");
            }
            else
            {
                _age = value;
            }
        }

        get
        {
            return _age;
        }
    }

       public Person()
       {
        Name = "Someone";
        Age = 21;
       }

    public string DescribeYourself()
    {
        string strDescription;

        strDescription = "My name is " + Name + " and I am " + Age + " years old";

        return strDescription;
    }

    public string DescribeYourself(string somebody)
    {
        string strDescription;

        strDescription = "Hello " + somebody + " My name is " + Name + " and I am " + Age + " years
old";

        return strDescription;
    }
}
```

Testing the constructor:

Create a page with a label called lblMessage. Add the following code to the code behind file.

```
Person Jane;
Jane = new Person();
```

```
lblMessage.Text = Jane.DescribeYourself();

lblMessage.Text += "<br />" + Jane.DescribeYourself("John");
```

Constructors can have parameters and can be overloaded.

### Activity 9. Overloading contructors

Modify the person class to contain a second constructor:

```
public Person()
{
        Name = "Someone";
        Age = 21;
}

public Person(string name, int age)
{
        Name = name;
        Age = age;
}
```

Like overloading methods the compiler will decide which on to execute depending on what is between the brackets.

You cannot call a constructor from a method but you can call a constructor from another constructor. For example we can change the parameter less constructor to call the other constructor.

```
public Person() : this("someone", 21)
{
}
```

If we look back at the student and instructor classes who both inherit from the person class. There is a constructor for person and there is one for student and instructor. When we create a new instance of a student first the parameter less constructor for person is executed and then the constructor for student is executed. Sometimes you may want to call a different constructor than the parameter less constructor of the base (or parent) class.

You can do this by using this syntax:

```
public Student(string name, int age)
        : base(name, age)
{
}
```

## Method and property hiding and overriding

Sometimes we need the derived class to override the method in the base class. We can do this in 2 ways by using hiding or overriding.

### Hiding example:

If this method is added to the student class, note the use of the “new” keyword. It will hide the base class method and implement the derived class method.

```
public new string DescribeYourself()
{
        return "Hi I am " + Name + " and I am a student";
}
```

So now when a student describes themselves the above code will be executed instead of the method in the Person class.

For this to work the method signature (the method name and input arguments) needs to be identical to the signature in the base class, otherwise it will be treated as an overloaded method.

### Overriding example:

When using overriding we need to modify the person class and the student class. The person class needs to have the keyword “virtual” added:

```
public virtual string  DescribeYourself()
{
        string strDescription;

        strDescription = "My name is " + Name + " and I am " + Age + " years old";

        return strDescription;
```

```
}
```

The method in the derived class then needs to use the “override” keyword to override the method:

```
public override string DescribeYourself()
{
        return "Hi I am " + Name + " and I am a student";
}
```