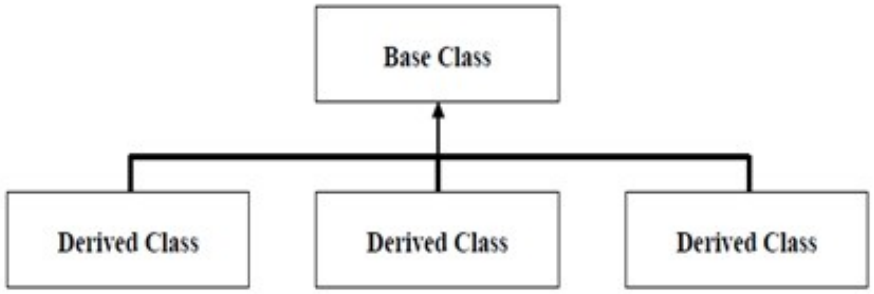
INHERITANCE AND POLYMORPHISM

Inheritance

Inheritance is a process of constructing or designing one class from another and is used to achieve extensibility. Extensibility of a module is it's potential to be extended (enhanced) as new needs evolve and is achieved by sub-classing and supporting the concept of hierarchical classification

Inheritance is one of the pillars of object-orientation. It is the mechanism of designing one class from another and is one of the ideas for code reusability. Reusability is the property of a module (a component, class or even a method) that enables it to be used in different applications without any or little change in its source code.

The original class (or the class that is sub-typed) is called the base, parent or super class. The class that inherits the functionality of the base class and extends it in its own way is called the sub, child, derived or inherited class.



If a class B (sub class) inherits a class A (base class), then B would have a copy of all the instance members (fields, methods, properties) of class A and B can access all the members (except for the private members) of class A. Private members of a base class do get inherited in a sub-class, but they cannot be accessed by the sub-class. Also, inheritance is said to create a, 'type of' relationship among classes which means sub-classes are a type of base class

Forms of Inheritance

i). Classical form (Also referred to as ‘is-a’ relationship) e.g. mammal is-a type of animal
ii). Containment form (Also referred to as containership inheritance) e.g. Car has-a radio

Types of Inheritance

i). Implementation inheritance derives from a base type, taking all the base type’s member fields and functions. With implementation inheritance, a derived type adopts the base type’s implementation of each function, unless it is indicated in the definition of the derived type that a function implementation is to be overridden. This type of inheritance is most useful when you need to add functionality to an existing type, or where a number of related types share a significant amount of common functionality.

ii). Interface inheritance inherits only the signatures of the functions, but does not inherit any implementations. This type of inheritance is most useful when you want to specify that a type makes certain features available. Interface inheritance is often regarded as providing a contract: By deriving from an interface, a type is guaranteed to provide certain functionality to clients.

Characteristic of Inheritance

- A derived class extends its direct base class. It can add new members to those it inherits. However, it cannot change or remove the definition of an inherited member.
- A derived class can hide an inherited member.
- A derived class can override an inherited member.
- An instance of a class contains a copy of all instance fields declared in the class and its base class.

Constructors and destructors are not inherited

Accessibility Constraints

Constraints on the accessibility of members and classes when they are used in process of inheritance:

i). The direct base class of a derived class must be at least as accessible as the derived class itself.
ii). Accessibility domain of a member is never larger than that of the class containing it.
iii). The return type of a method must be as accessible as the method itself.

Inheritance in C#

The following are some key-points about inheritance in C#

- C#, like Java and contrary to C++, allows only single class inheritance. Multiple inheritance of classes is not allowed in C#.
- The Object class defined in the System namespace is implicitly the ultimate base class of all the classes in C# (and the .NET framework)

The multiple inheritance of interfaces is allowed in C# (similar to Java). Structures (struct) in C# can only inherit (or implement) interfaces and cannot be inherited.

Implementing inheritance in C#

C# uses the colon ':' operator to indicate inheritance.

Syntax

```
class <name> : <baseclass name> {}
```

Example

```
class Employee {
     //members of the Employee class
}
class Manager : Employee {
     //members of the Manager class
}
```

The Manager class inherits the Employee class by using the colon operator.

```
class Manager : Employee
```

The Manager class inherits all the members of the Employee class. In addition, it also declares its own members Notice C# does not support private inheritance, hence the absence of a public or private qualifier on the base class name

Constructor calls in Inheritance

Constructors are called in order of System.Object first, then progressing down the hierarchy until the compiler reaches the class being instantiated. Also each constructor handles initialization of the fields in its own class. That's how it should normally work, and when you start adding your own constructors you should try to stick to that principle.

The base keyword - Calling Constructors of the base-class explicitly

We can explicitly call the constructor of a base-class using the keyword base. base must be used with the constructor header (or signature or declaration) after a colon ':' operator as follows:

```
class SubClass : BaseClass
{
     SubClass(int id) : base() // explicit constructor call
     {
          // some code goes here
```

```
        }
        //or
        SubClass(int id) // implicit constructor call
        {
            // some code goes here
        }

    }
```

We can also call the parameterized constructor of the base-class through the base keyword, as shown below:

```
    class SubClass : BaseClass
    {
        SubClass(int id) : base(id)
        {
            // some code goes here
        }
    }
```

Now, the constructor of SubClass(int) will explicitly call the constructor of the base-class (BaseClass) that takes an int argument, delegating the initialization of the inherited fields to the base-class's parameterized constructor.

Notice that C#'s base keyword is similar to Java's super keyword

Constructor Initializers

The base and this keywords are the only keywords allowed in the line which calls another constructor. Anything else causes a compilation error. These constructor initializers enable you to specify which class and which constructor you want called. This takes two forms:

i). An initializer of the form base() enables the current class's base class constructor i.e. the specific constructor implied by the form of the constructor called to be called.

ii). An initializer taking the form this() enables the current class to call another constructor defined within itself. This is useful when you have overloaded multiple constructors and want to make sure that a default constructor is always called.

```
public class Employee
{
  public Employee ()
  {
      …
  }
  public Employee (int n) : this ()
  {
      …
  }
}
```

Protected Access Modifier

The protected members of the class can be accessed either inside the containing class or inside its sub-class.

Example

```
    using System;
    class Rectangle
    {
        protected int length;
        protected int breadth;
```

```
        public Rectangle(int length, int breadth)
        {
              this.length=length;
              this.breadth=breadth;
        }
        public int Area()
        {
              return(length*breadth);
        }
  }
  class Cubicle:Rectangle
  {
        int height;
        public    Cubicle(int    length,int    breadth,int
        height) :base(length,breadth)
        {
              this.height=height;
        }
        public int Volume()
        {
              return(length*breadth*height);
        }
  }
  class InherTest
  {
        public static void Main()
        {
              Cubicle cubicle1=new Cubicle(10,11,12);
              Console.WriteLine("Area= "+cubicle1.Area());
              Console.WriteLine("Volume=  "+cubicle1.Volume());
        }
  }
```

The Protected internal Access Modifier

In a similar way, the protected internal access modifier allows a member (field, property and method) to be accessed:

- Inside the containing class, or
- Inside the same project, or
- Inside the sub-class of the containing class.

Hence, protected internal acts like 'protected OR internal', i.e., either protected or internal.

The sealed keyword

Prevent a class from being further sub classed. Such classes are called sealed classes and no class can inherit from a sealed class. Standalone utility classes are created as sealed classes.

Example:

```
sealed class AClass
{
     …..
}
sealed class BClass:Someclass
```

```
{
        ….
}
```
Any attempt to inherit these classes will cause an error and compiler will not allow it.
Notice that C#'s sealed keyword is identical to Java's final keyword when applied to classes.

Sealed Methods

When an instance method declaration includes the sealed modifier, the method is said to be a sealed method. A derived class cannot override this method. A sealed method is used to override an inherited virtual method with the same signature.

Example:

```
class A
{
        public virtual void Fun()
        {
                ……
        }
}
class B: A
{
        public sealed override void Fun()
        {
                ……
        }
}
```
Now any derived class of B cannot further override the method Fun().

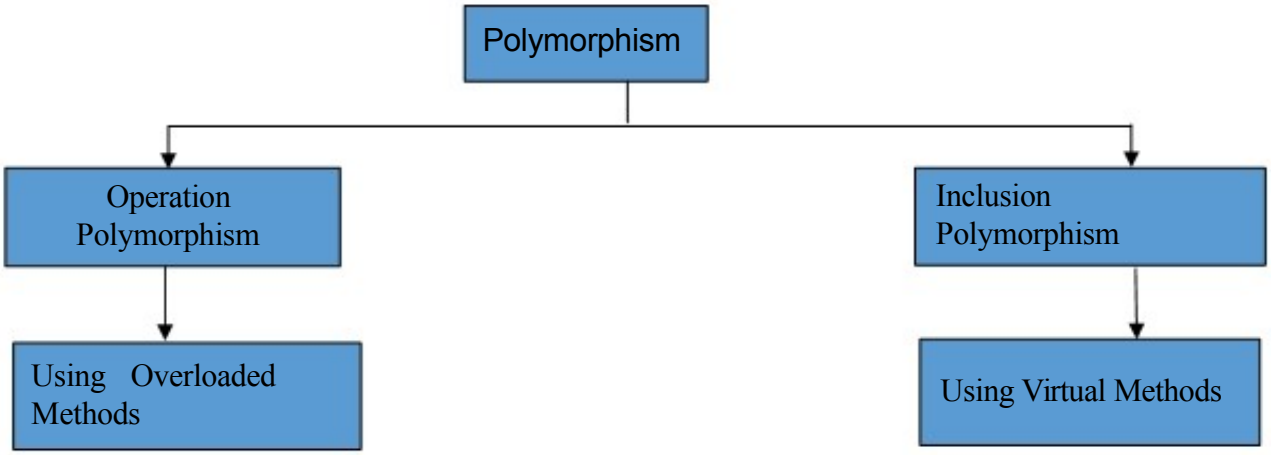
Object class - the base of all classes

In C# (and the .NET framework) all the types (classes, structures and interfaces) are implicitly inherited from the Object class defined in the System namespace. This class provides the low-level services and general functionality to each and every class in the .NET framework. The Object class is extremely useful when it comes to polymorphism, as a reference of type Object can hold any type of object. C# also provides an object keyword which maps to the System.Object class in the .NET framework class library (FCL).
If you do not specify a base class in a class definition, the C# compiler will assume that System.Object is the base class.

```
class MyClass : Object // derives from System.Object
{
      // etc.
}
```
Is same as:
```
class MyClass // derives from System.Object
{
      // etc.
}
```
For the sake of simplicity, the second form is more common.

Polymorphism

Polymorphism is the ability for classes to provide different implementations of methods that are called by the same name. Polymorphism allows a method of a class to be called without regard to what specific implementation it provides. Polymorphism can be achieved in two ways:



i). Operation Polymorphism (compile time/static polymorphism)

It is also referred as compile time polymorphism and is implemented using overloaded methods and operators. Overloaded methods are selected for invoking by matching arguments, in terms of number, type and order, at compile time. It is also called as early binding, or static binding or static linking.

ii). Inclusion Polymorphism (run-time/dynamic polymorphism)

It is also referred as run-time polymorphism and is achieved through use of virtual functions.
Since method is linked with a particular class much later after compilation, this process is also called late binding. Also known as dynamic binding as the selection of method is done dynamically at run time.

References and Inheritance

An object reference can refer to an object of its class, or to an object of any class derived from it by inheritance. For example, if the Shape class is used to derive a child class called Rectangle, then a Shape reference can be used to point to a Rectangle object.

```
Shape shape
shape = new Shape();
…
shape = new Rectangle();
```

Overriding the methods - virtual and override methods

A class can declare virtual methods, properties, and indexers, and derived classes can override the implementation of these function members. This enables classes to exhibit polymorphic behavior wherein the actions performed by a function member invocation vary depending on the run-time type of the instance through which the function member is invoked.
When you override a virtual method in a derived class, you need to use the override keyword to signal that you are overriding a virtual method. virtual methods must be used on the base class's method, and the override methods are used on the derived class's implementation of the method.
Polymorphism requires different implementations of a method with the same name and signature in the base and sub-classes. When such a method is called using a base-type reference, the compiler uses the actual object type referenced by the base type reference to decide which of the methods to call. Notice that neither member fields nor static functions can be declared as virtual.

Example

If we want to override the Area() method of the Shape class in the Circle class, we have to mark the Area() method in the Shape (base) class as virtual and the Area() method in the Circle (sub) class as override.

```
class Shape
{
     public virtual void Area()
     {
          Console.WriteLine("Area of Shape...");

     }
}
class Circle : Shape
{
     public override void Area()
     {
          Console.WriteLine("Area of Circle...");
     }
}
```

Now, in our Main() method we write

```
static void Main()
{
     Shape theShape = new Circle();
     theShape.Area();
}
```

Here we have used the reference of the base type (Shape) to refer to an object of the sub type (Circle) and call the Area() method through it. As we have overridden the Area() method of Shape in the Circle class and since the Area() method is marked virtual in Shape, the compiler will no longer see the apparent (or reference) type to call the method (static, early or compile time object binding), rather, it will apply "dynamic, late or runtime object binding" and will see the object type at 'runtime' to decide which Area() method it should call. When we compile the above program, the result will be:

```
Area of Circle...
```

Although, we called the Area() method using the reference of Shape type, the CLR will consider the object held by the Shape reference and calls the Area() method of the Circle class.

Note: All the methods in C# are non-virtual by default unlike Java (where all the methods are implicitly virtual). We have to explicitly mark a method as virtual (like C++). Unlike C++ and Java, we also have to declare the overriding method in the sub-class as override in order to avoid any unconscious overriding

Complete Example

```
using System;
class Shape
{
     public virtual void Area()
     {
          Console.WriteLine("Area of Shape...");
     }
}
class Circle:Shape
{
```

```
     private double radius;
     public Circle(double radius)
     {
          this.radius=radius;
     }
     public override void Area()
     {
          Console.WriteLine("Area of Circle " +
          Math.PI*Math.Pow(radius,2));
      }
}
class Rectangle:Shape
{
     private double length, width;
     public Rectangle(double length, double width)
     {
          this.length=length;
          this.width=width;  }
     public override void Area()
     {
          Console.WriteLine("Area of Rectangle " +
          length*width);
     }
}
class Triangle:Shape
{
     private double tBase, tHeight;
     public Triangle(double tBase, double tHeight)
     {
          this.tBase=tBase;
          this.tHeight=tHeight;
     }
     public override void Area()
     {
          Console.WriteLine("Area of Triangle " +
          1.0/2.0*tBase*tHeight);
     }
}
class Tester
{
     static void Main()
     {
          Shape [] shapes = {new Circle(7), new Rectangle(5,4), new
          Triangle(10,5)};
          Random random = new Random();
          for(int i=0; i<5; i++)
          {
               int randNum = random.Next(0, 3);
               shapes[randNum].Area();
          }
     }
}
```

Property as virtual

For a virtual or overridden property, the syntax is the same as for a non-virtual property with the exception of the keyword virtual, which is added to the definition.

Syntax:

```
public virtual string FirstName{
     get { return firstName;}
     set { firstName = value;}
}

private string firstName;
```

Leverage base class members

A derived class that has replaced or overridden a method or property can still access the method or property on the base class using the base keyword

```
     public class BaseClass
     {
          public virtual void DoWork ()
          {
               …
          }
          …
     }
     public class DerivedClass : BaseClass
     {
          public override void DoWork ()
          {
               base.DoWork();
          }
     }
```

The new keyword

The new modifier is used to explicitly hide a member inherited from a base. Generally, the keyword new is used to mark a method as a non-overriding method and which cannot be used polymorphically. Hidden class members can still be called if an instance of the derived class is cast to an instance of the base class

Example

```
 class Shape
 {
     public virtual void Area()
     {
         Console.WriteLine("Area Shape...");
     }
 }
 class Circle : Shape
 {
     public new void Area()
     {
         Console.WriteLine("Area Circle...");
     }
 }
```

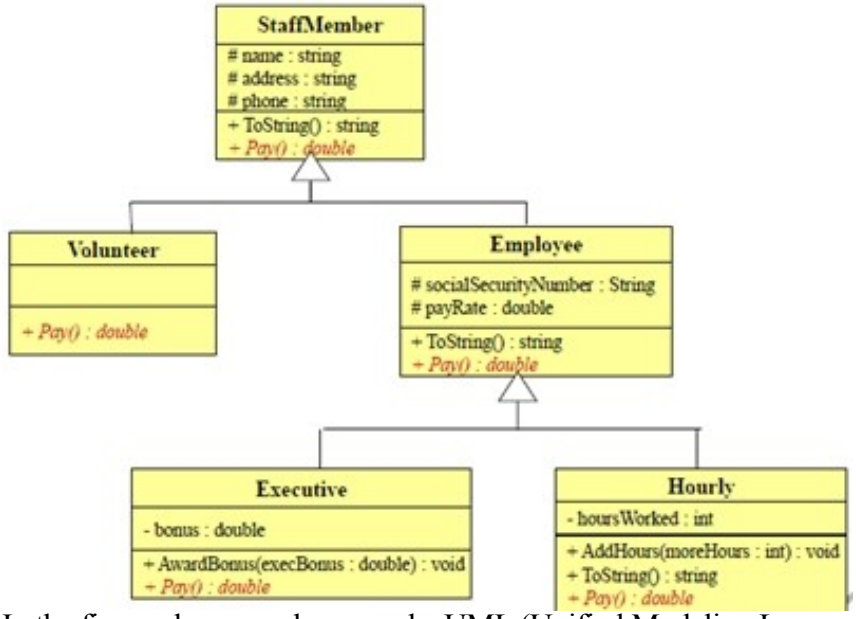
Note that we marked the Area() method in Circle with the new keyword to avoid polymorphism. To make the current member override that implementation, add the override keyword. Otherwise add the new keyword.

Difference between new and override

- The new members are the members of derived class, while override members are the members of the base class and can be redefined in derived class
- C# allows a subclass' member to have the same name with that of the base class. In this case, the member in subclass is seemed as a new member

UML Notation

The following is a an illustration of the UML notation in the design process



In the figure above, we have used a UML (Unified Modeling Language) class diagram. Here, StaffMember is the overall base class. A base class usually has general functionality, while sub-classes possess specific functionality.

Type casting the objects: Up-casting and Down-casting

Type Casting means 'making an object behave like' or 'changing the apparent type of an object'. In C#, you can cast an object in an inheritance hierarchy either from the bottom towards the top (up-casting) or from the top towards the bottom (down-casting).

Up-casting is simple, safe and implicit, as we have seen the reference of parent type can reference the object of child type.

```
Parent theParent = new Child();
```

On the contrary, down-casting is un-safe and explicit i.e. it may throw an exception (fail). Consider the following line of code:

```
Shape [] shapes = {new Circle(), new Rectangle(), new Triangle()};
```

Where Circle, Rectangle and Triangle are sub-classes of Shape class. Now, if we want to reference the Rectangle object in the shapes array with the reference of type Rectangle, we can't just write

```
Rectangle rect = shapes[1];
```

Instead, we have to explicitly apply the cast here as

```
Rectangle rect = (Rectangle) shapes[1];
```

Also, note that down-casting can be unsuccessful and if we attempt to write

```
Rectangle rect = (Rectangle) shapes[2];
```

Since shapes[2] contains an object of type Triangle which cannot be casted to the Rectangle type

The is and as Operators

i). The is Operator

The is operator allows us to check whether an object is compatible with a specific type i.e. object is either of that type or is derived from that type. Generally, is compares the type of the object with the given type and returns true if it is cast-able; otherwise, it returns false. For example,

```
Console.WriteLine(shapes[1] is Rectangle);
```

would print true on the Console Window, while

```
Console.WriteLine(shapes[2] is Rectangle);
```

would print false on the Console window. We might use the is operator to check the Runtime Type Checks of an object before applying down-casting.

```
Shape [] shapes = {new Circle(), new Rectangle(), new Curve()};
Rectangle rect=null;
if(shapes[1] is Rectangle)
{
     rect = (Rectangle) shapes[1];
}
```

ii). The as Operator

The as operator is used to perform explicit type conversions of reference types. If the type being converted is compatible with the specified type, conversion is performed successfully. However, if the types are incompatible, then the as operator returns the value null.
Although is and as perform similar functionality, is just checks the runtime type while as in addition to this, also performs Checked Type Casts as shown below:

```
Shape [] shapes = {new Circle(7), new Rectangle(5,4), new Triangle(10,5)};
Rectangle rect = shapes[1] as Rectangle;
if(rect != null)
     Console.WriteLine("Cast successful");
else
     Console.WriteLine("Cast unsuccessful");
```

The advantage of using the as operator allows you to perform a safe type conversion in a single step without the need to first test the type using the is operator and then perform the conversion.

Nested Classes (Inner Classes) in C#

Nested classes (also called inner classes) are defined inside another class, as shown below:

```
class Outer
{
     private static string name="My Name";
     public Outer()
     {
          Console.WriteLine("Constructor of Outer called...");
```

```
    }
    public class Inner
    {
        public Inner()
        {
            Console.WriteLine("Constructor of Inner called...");
        }
        public void InnerDisplay()
        {
            Console.WriteLine("InnerDisplay() called...");
            Console.WriteLine("Name in Outer is {0}", name);
        }
    }
}
```

While the top-level classes can only be marked as public or internal, nested classes can be marked with all the access modifiers: private, public, protected, internal, internal OR protected. Like other class members, the default access modifier for nested classes is private. In order to reference and instantiate the Inner class in the Main() method of the Tester class, we mark it as public in the program. It is also worth-noting that nested classes are always accessed and instantiated with reference to their container or enclosing class as shown below:

```
    class Tester
    {
        static void Main()
        {
            Outer.Inner inner = new Outer.Inner();
            inner.InnerDisplay();
        }
    }
```

Another important point about nested classes is that they can access all the (private, internal, protected, public) static members of the enclosing class.
Since the nested classes are instantiated with reference to the enclosing class, their access protection level always remains less than or equal to that of the enclosing class. Hence, if the access level of the enclosing class is internal and the access level of the nested class is public, the nested class would only be accessible in the current assembly (project).