Unit II

OBJECT ORIENTED ASPECTS OF C#

Key Concepts of Object Orientation

- Abstraction
- Encapsulation
- Polymorphism
- Inheritance.

Abstraction is the ability to generalize an object as a data type that has a specific set of characteristics and is able to perform a set of actions.

Object-oriented languages provide abstraction via classes. Classes define the properties and methods of an object type.

Examples:

- You can create an abstraction of a dog with characteristics, such as color, height, and weight, and actions such as run and bite. The characteristics are called properties, and the actions are called methods.
- A Recordset object is an abstract representation of a set of data.

Classes

When you define a class, you define a blueprint for a data type. This does not actually define any data, but it does define what the class name means. That is, what an object of the class consists of and what operations can be performed on that object. Objects are instances of a class. The methods and variables that constitute a class are called members of the class.

Defining a Class

A class definition starts with the keyword class followed by the class name; and the class body enclosed by a pair of curly braces. Following is the general form of a class definition:

```
<access specifier> class  class_name
{
   // member variables
   <access specifier> <data type> variable1;
   <access specifier> <data type> variable2;
   ...
   <access specifier> <data type> variableN;
   // member methods
   <access specifier> <return type> method1(parameter_list)
   {
      // method body
   }
   <access specifier> <return type> method2(parameter_list)
   {
      // method body
   }
   ...
   <access specifier> <return type> methodN(parameter_list)
   {
      // method body
   }
```

```
}
```

Note:

- Access specifiers specify the access rules for the members as well as the class itself. If not mentioned, then the default access specifier for a class type is **internal**. Default access for the members is **private**.
- Data type specifies the type of variable, and return type specifies the data type of the data the method returns, if any.
- To access the class members, you use the dot (.) operator.
- The dot operator links the name of an object with the name of a member.

The following example illustrates the concepts discussed so far:

```
using System;
namespace BoxApplication
{
    class Box
    {
       public double length;   // Length of a box
       public double breadth;  // Breadth of a box
       public double height;   // Height of a box
    }
    class Boxtester
    {
        static void Main(string[] args)
        {
            Box Box1 = new Box();   // Declare Box1 of type Box
            Box Box2 = new Box();   // Declare Box2 of type Box
            double volume = 0.0;    // Store the volume of a box here

            // box 1 specification
            Box1.height = 5.0;
```

```
            Box1.length = 6.0;
            Box1.breadth = 7.0;

            // box 2 specification
            Box2.height = 10.0;
            Box2.length = 12.0;
            Box2.breadth = 13.0;

            // volume of box 1
            volume = Box1.height * Box1.length * Box1.breadth;
            Console.WriteLine("Volume of Box1 : {0}",  volume);

            // volume of box 2
            volume = Box2.height * Box2.length * Box2.breadth;
            Console.WriteLine("Volume of Box2 : {0}", volume);
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

Member Functions and Encapsulation

A member function of a class is a function that has its definition or its prototype within the class definition similar to any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Member variables are the attributes of an object (from design perspective) and they are kept private to implement encapsulation. These variables can only be accessed using the public member functions.

Let us put above concepts to set and get the value of different class members in a class:

```
using System;

namespace BoxApplication
{
   class Box
   {
      private double length;   // Length of a box
      private double breadth;  // Breadth of a box
      private double height;   // Height of a box
      public void setLength( double len )
      {
         length = len;
      }

      public void setBreadth( double bre )
      {
         breadth = bre;
      }

      public void setHeight( double hei )
      {
         height = hei;
      }
      public double getVolume()
      {
         return length * breadth * height;
      }
   }
   class Boxtester
   {
      static void Main(string[] args)
```

```
      {
         Box Box1 = new Box();   // Declare Box1 of type Box
         Box Box2 = new Box();
         double volume;

         // Declare Box2 of type Box
         // box 1 specification
         Box1.setLength(6.0);
         Box1.setBreadth(7.0);
         Box1.setHeight(5.0);

         // box 2 specification
         Box2.setLength(12.0);
         Box2.setBreadth(13.0);
         Box2.setHeight(10.0);

         // volume of box 1
         volume = Box1.getVolume();
         Console.WriteLine("Volume of Box1 : {0}" ,volume);

         // volume of box 2
         volume = Box2.getVolume();
         Console.WriteLine("Volume of Box2 : {0}", volume);

         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

C# Constructors

A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.

A constructor has exactly the same name as that of class and it does not have any return type. Following example explains the concept of constructor:

```
using System;

namespace LineApplication

{

   class Line

   {

      private double length;   // Length of a line

      public Line()

      {

         Console.WriteLine("Object is being created");

      }


      public void setLength( double len )

      {

         length = len;

      }


      public double getLength()

      {

         return length;

      }


      static void Main(string[] args)

      {

         Line line = new Line();


         // set line length
```

```
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created
Length of line : 6
```

A **default constructor** does not have any parameter but if you need, a constructor can have parameters. Such constructors are called **parameterized constructors**. This technique helps you to assign initial value to an object at the time of its creation as shown in the following example:

```
using System;
namespace LineApplication
{
   class Line
   {
      private double length;   // Length of a line
      public Line(double len)  //Parameterized constructor
      {
         Console.WriteLine("Object is being created, length = {0}", len);
         length = len;
      }

      public void setLength( double len )
      {
         length = len;
      }
      public double getLength()
```

```
      {
         return length;
      }

      static void Main(string[] args)
      {
         Line line = new Line(10.0);
         Console.WriteLine("Length of line : {0}", line.getLength());

         // set line length
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

C# Destructors

A **destructor** is a special member function of a class that is executed whenever an object of its class goes out of scope. A **destructor** has exactly the same name as that of the class with a prefixed tilde (~) and it can neither return a value nor can it take any parameters.

Destructor can be very useful for releasing memory resources before exiting the program. Destructors cannot be inherited or overloaded.

Following example explains the concept of destructor:

```
using System;
```

```
namespace LineApplication
{
   class Line
   {
      private double length;   // Length of a line
      public Line()  // constructor
      {
         Console.WriteLine("Object is being created");
      }
      ~Line() //destructor
      {
         Console.WriteLine("Object is being deleted");
      }

      public void setLength( double len )
      {
         length = len;
      }

      public double getLength()
      {
         return length;
      }

      static void Main(string[] args)
      {
         Line line = new Line();

         // set line length
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
      }
   }
```

```
}
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created
Length of line : 6
Object is being deleted
```

Static Members of a C# Class

We can define class members as static using the **static** keyword. When we declare a member of a class as static, it means no matter how many objects of the class are created, there is only one copy of the static member.

The keyword **static** implies that only one instance of the member exists for a class. Static variables are used for defining constants because their values can be retrieved by invoking the class without creating an instance of it. Static variables can be initialized outside the member function or class definition. You can also initialize static variables inside the class definition.

The following example demonstrates the use of **static variables**:

```
using System;

namespace StaticVarApplication

{

   class StaticVar

   {

      public static int num;

      public void count()

      {

         num++;

      }

      public int getNum()

      {

         return num;
```

```
      }
   }
   class StaticTester
   {
      static void Main(string[] args)
      {
         StaticVar s1 = new StaticVar();
         StaticVar s2 = new StaticVar();
         s1.count();
         s1.count();
         s1.count();
         s2.count();
         s2.count();
         s2.count();
         Console.WriteLine("Variable num for s1: {0}", s1.getNum());
         Console.WriteLine("Variable num for s2: {0}", s2.getNum());
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Variable num for s1: 6
Variable num for s2: 6
```

You can also declare a **member function** as **static**. Such functions can access only static variables. The static functions exist even before the object is created. The following example demonstrates the use of **static functions**:

```
using System;
namespace StaticVarApplication
{
```

```
   class StaticVar
   {
      public static int num;
      public void count()
      {
         num++;
      }
      public static int getNum()
      {
         return num;
      }
   }
   class StaticTester
   {
      static void Main(string[] args)
      {
         StaticVar s = new StaticVar();
         s.count();
         s.count();
         s.count();
         Console.WriteLine("Variable num: {0}", StaticVar.getNum());
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Variable num: 3
```

C# - Inheritance

One of the most important concepts in object-oriented programming is inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and speeds up implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base**class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **IS-A** relationship. For example, mammal**IS A** animal, dog **IS-A** mammal hence dog **IS-A** animal as well, and so on.

Base and Derived Classes

A class can be derived from more than one class or interface, which means that it can inherit data and functions from multiple base classes or interfaces.

The syntax used in C# for creating derived classes is as follows:

```
<acess-specifier> class <base_class>
{
   ...
}
class <derived_class> : <base_class>
{
   ...
}
```

Consider a base class Shape and its derived class Rectangle:

```
using System;

namespace InheritanceApplication
```

```
{
   class Shape
   {
      public void setWidth(int w)
      {
         width = w;
      }
      public void setHeight(int h)
      {
         height = h;
      }
      protected int width;
      protected int height;
   }

   // Derived class
   class Rectangle: Shape
   {
      public int getArea()
      {
         return (width * height);
      }
   }

   class RectangleTester
   {
      static void Main(string[] args)
      {
         Rectangle Rect = new Rectangle();

         Rect.setWidth(5);
         Rect.setHeight(7);
```

```
         // Print the area of the object.
         Console.WriteLine("Total area: {0}",  Rect.getArea());
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Total area: 35
```

Initializing Base Class

The derived class inherits the base class member variables and member methods. Therefore the super class object should be created before the subclass is created. You can give instructions for superclass initialization in the member initialization list.

The following program demonstrates this:

```
using System;
namespace RectangleApplication
{
   class Rectangle
   {
      //member variables
      protected double length;
      protected double width;
      public Rectangle(double l, double w)
      {
         length = l;
         width = w;
      }

      public double GetArea()
```

```
    {
        return length * width;
    }

    public void Display()
    {
        Console.WriteLine("Length: {0}", length);
        Console.WriteLine("Width: {0}", width);
        Console.WriteLine("Area: {0}", GetArea());
    }
}//end class Rectangle

class Tabletop : Rectangle
{
    private double cost;
    public Tabletop(double l, double w) : base(l, w)
    { }
    public double GetCost()
    {
        double cost;
        cost = GetArea() * 70;
        return cost;
    }
    public void Display()
    {
        base.Display();
        Console.WriteLine("Cost: {0}", GetCost());
    }
}
class ExecuteRectangle
{
    static void Main(string[] args)
    {
```

```
         Tabletop t = new Tabletop(4.5, 7.5);
         t.Display();
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Length: 4.5
Width: 7.5
Area: 33.75
Cost: 2362.5
```

Multiple Inheritance in C#

C# does not support multiple inheritance. However, you can use interfaces to implement multiple inheritance. The following program demonstrates this:

```
using System;
namespace InheritanceApplication
{
   class Shape
   {
      public void setWidth(int w)
      {
         width = w;
      }
      public void setHeight(int h)
      {
         height = h;
      }
      protected int width;
      protected int height;
```

```
}

// Base class PaintCost
public interface PaintCost
{
   int getCost(int area);

}

// Derived class
class Rectangle : Shape, PaintCost
{
   public int getArea()
   {
      return (width * height);
   }
   public int getCost(int area)
   {
      return area * 70;
   }
}
class RectangleTester
{
   static void Main(string[] args)
   {
      Rectangle Rect = new Rectangle();
      int area;
      Rect.setWidth(5);
      Rect.setHeight(7);
      area = Rect.getArea();

      // Print the area of the object.
      Console.WriteLine("Total area: {0}",  Rect.getArea());
```

```
            Console.WriteLine("Total paint cost: ${0}" , Rect.getCost(area));
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Total area: 35
Total paint cost: $2450
```

Understanding Properties in C#

In C#, properties are nothing but natural extension of data fields. They are usually known as 'smart fields' in C# community. We know that data encapsulation and hiding are the two fundamental characteristics of any object oriented programming language.In C#, data encapsulation is possible through either classes or structures. By using various access modifiers like private, public, protected, internal etc it is possible to control the accessibility of the class members.

Usually inside a class, we declare a data field as private and will provide a set of public SET and GET methods to access the data fields. This is a good programming practice, since the data fields are not directly accessible out side the class. We must use the set/get methods to access the data fields.

An example, which uses a set of set/get methods, is shown below.

```
//SET/GET methods
//Author: rajeshvs@msn.com
using System;
class MyClass
{
private int x;
public void SetX(int i)
{
```

```
x = i;
}
public int GetX()
{
return x;
}
}
class MyClient
{
public static void Main()
{
MyClass mc = new MyClass();
mc.SetX(10);
int xVal = mc.GetX();
Console.WriteLine(xVal);//Displays 10
}
}
```

But C# provides a built in mechanism called properties to do the above. In C#, properties are defined using the property declaration syntax. The general form of declaring a property is as follows.

```
<acces_modifier> <return_type> <property_name>
{
get
```

```
{
}
set
{
}
}
```

Where <access_modifier> can be private, public, protected or internal. The <return_type> can be any valid C# type. Note that the first part of the syntax looks quite similar to a field declaration and second part consists of a get accessor and a set accessor.

For example the above program can be modifies with a property X as follows.

```
class MyClass
{
private int x;
public int X
{
get
{
return x;
}
set
```

```
{
x = value;
}
}
}
```

The object of the class MyClass can access the property X as follows.

```
MyClass mc = new MyClass();
```

mc.X = 10; // calls set accessor of the property X, and pass 10 as value of the standard field 'value'.
This is used for setting value for the data member x.
Console.WriteLine(mc.X);// displays 10. Calls the get accessor of the property X.

The complete program is shown below.

```
//C#: Property
//Author: rajeshvs@msn.com
using System;
class MyClass
{
private int x;
public int X
{
```

```
get
{
return x;
}
set
{
x = value;
}
}
}
class MyClient
{
public static void Main()
{
MyClass mc = new MyClass();
mc.X = 10;
int xVal = mc.X;
Console.WriteLine(xVal);//Displays 10
}
}
```

Remember that a property should have at least one accessor, either set or get. The set accessor has a free variable available in it called value, which

gets created automatically by the compiler. We can't declare any variable with the name value inside the set accessor.

We can do very complicated calculations inside the set or get accessor. Even they can throw exceptions.

Since normal data fields and properties are stored in the same memory space, in C#, it is not possible to declare a field and property with the same name.

Static Properties

C# also supports static properties, which belongs to the class rather than to the objects of the class. All the rules applicable to a static member are applicable to static properties also.

The following program shows a class with a static property.

```
//C# : static Property
//Author: rajeshvs@msn.com
using System;
class MyClass
{
private static int x;
public static int X
{
get
```

```
{
return x;
}
set
{
x = value;
}
}
}
class MyClient
{
public static void Main()
{
MyClass.X = 10;
int xVal = MyClass.X;
Console.WriteLine(xVal);//Displays 10
}
}
```

Remember that set/get accessor of static property can access only other static members of the class. Also static properties are invoking by using the class name.

Properties & Inheritance

The properties of a Base class can be inherited to a Derived class.

```
//C# : Property : Inheritance
//Author: rajeshvs@msn.com
using System;
class Base
{
public int X
{
get
{
Console.Write("Base GET");
return 10;
}
set
{
Console.Write("Base SET");
}
}
}
class Derived : Base
{
}
class MyClient
```

```
{
public static void Main()
{
Derived d1 = new Derived();
d1.X = 10;
Console.WriteLine(d1.X);//Displays 'Base SET Base GET 10'
}
}
```

The above program is very straightforward. The inheritance of properties is just like inheritance any other member.

Properties & Polymorphism

A Base class property can be polymorphicaly overridden in a Derived class. But remember that the modifiers like virtual, override etc are using at property level, not at accessor level.

```
//C# : Property : Polymorphism
//Author: rajeshvs@msn.com
using System;
class Base
{
public virtual int X
{
```

```
get
{
Console.Write("Base GET");
return 10;
}
set
{
Console.Write("Base SET");
}
}
}
class Derived : Base
{
public override int X
{
get
{
Console.Write("Derived GET");
return 10;
}
set
{
Console.Write("Derived SET");
}
```

```
}
}
class MyClient
{
public static void Main()
{
Base b1 = new Derived();
b1.X = 10;
Console.WriteLine(b1.X);//Displays 'Derived SET Derived GET 10'
}
}
```

Abstract Properties

A property inside a class can be declared as abstract by using the keyword abstract. Remember that an abstract property in a class carries no code at all. The get/set accessors are simply represented with a semicolon. In the derived class we must implement both set and get assessors.

If the abstract class contains only set accessor, we can implement only set in the derived class.

The following program shows an abstract property in action.

```
//C# : Property : Abstract
```

```
//Author: rajeshvs@msn.com
using System;
abstract class Abstract
{
public abstract int X
{
get;
set;
}
}
class Concrete : Abstract
{
public override int X
{
get
{
Console.Write(" GET");
return 10;
}
set
{
Console.Write(" SET");
}
}
```

```
}
class MyClient
{
public static void Main()
{
Concrete c1 = new Concrete();
c1.X = 10;
Console.WriteLine(c1.X);//Displays 'SET GET 10'
}
}
```

The properties are an important features added in language level inside C#. They are very useful in GUI programming. Remember that the compiler actually generates the appropriate getter and setter methods when it parses the C# property syntax.

C# - Indexers

An **indexer** allows an object to be indexed such as an array. When you define an indexer for a class, this class behaves similar to a **virtual array**. You can then access the instance of this class using the array access operator ([ ]).

Syntax

A one dimensional indexer has the following syntax:

```
element-type this[int index]
{
   // The get accessor.
   get
   {
      // return the value specified by index
   }

   // The set accessor.
   set
   {
      // set the value specified by index
```

```
    }
}
```

Use of Indexers

Declaration of behavior of an indexer is to some extent similar to a property. similar to the properties, you use **get** and **set** accessors for defining an indexer. However, properties return or set a specific data member, whereas indexers returns or sets a particular value from the object instance. In other words, it breaks the instance data into smaller parts and indexes each part, gets or sets each part.

Defining a property involves providing a property name. Indexers are not defined with names, but with the **this** keyword, which refers to the object instance. The following example demonstrates the concept:

```
using System;
namespace IndexerApplication
{
   class IndexedNames
   {
      private string[] namelist = new string[size];
      static public int size = 10;
```

```
public IndexedNames()
{
   for (int i = 0; i < size; i++)
   namelist[i] = "N. A.";
}

public string this[int index]
{
   get
   {
      string tmp;

      if( index >= 0 && index <= size-1 )
      {
         tmp = namelist[index];
      }
      else
      {
         tmp = "";
      }
```

```
            return ( tmp );
         }
         set
         {
            if( index >= 0 && index <= size-1 )
            {
               namelist[index] = value;
            }
         }
      }

      static void Main(string[] args)
      {
         IndexedNames names = new IndexedNames();
         names[0] = "Zara";
         names[1] = "Riz";
         names[2] = "Nuha";
         names[3] = "Asif";
         names[4] = "Davinder";
```

```
            names[5] = "Sunil";
            names[6] = "Rubic";
            for ( int i = 0; i < IndexedNames.size; i++ )
            {
               Console.WriteLine(names[i]);
            }

            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Zara
Riz
Nuha
Asif
Davinder
Sunil
Rubic
```

```
N. A.
N. A.
N. A.
```

Overloaded Indexers

Indexers can be overloaded. Indexers can also be declared with multiple parameters and each parameter may be a different type. It is not necessary that the indexes have to be integers. C# allows indexes to be of other types, for example, a string.

The following example demonstrates overloaded indexers:

```
using System;
namespace IndexerApplication
{
   class IndexedNames
   {
      private string[] namelist = new string[size];
      static public int size = 10;
      public IndexedNames()
      {
         for (int i = 0; i < size; i++)
```

```
        {
            namelist[i] = "N. A.";
        }
    }

    public string this[int index]
    {
        get
        {
            string tmp;

            if( index >= 0 && index <= size-1 )
            {
                tmp = namelist[index];
            }
            else
            {
                tmp = "";
            }
```

```
            return ( tmp );
        }
        set
        {
            if( index >= 0 && index <= size-1 )
            {
                namelist[index] = value;
            }
        }
    }
    public int this[string name]
    {
        get
        {
            int index = 0;
            while(index < size)
            {
                if (namelist[index] == name)
                {
                 return index;
```

```
            }
            index++;
        }
        return index;
    }

}

static void Main(string[] args)
{
    IndexedNames names = new IndexedNames();
    names[0] = "Zara";
    names[1] = "Riz";
    names[2] = "Nuha";
    names[3] = "Asif";
    names[4] = "Davinder";
    names[5] = "Sunil";
    names[6] = "Rubic";

    //using the first indexer with int parameter
```

```
            for (int i = 0; i < IndexedNames.size; i++)
            {
                Console.WriteLine(names[i]);
            }

            //using the second indexer with the string
parameter
            Console.WriteLine(names["Nuha"]);
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Zara
Riz
Nuha
Asif
Davinder
Sunil
```

```
Rubic
N. A.
N. A.
N. A.
2
```

C# - Polymorphism

The word **polymorphism** means having many forms. In object-oriented programming paradigm, polymorphism is often expressed as 'one interface, multiple functions'.

Polymorphism can be static or dynamic. In **static polymorphism**, the response to a function is determined at the compile time. In **dynamic polymorphism**, it is decided at run-time.

Static Polymorphism

The mechanism of linking a function with an object during compile time is called early binding. It is also called static binding. C# provides two techniques to implement static polymorphism. They are:

- Function overloading
- Operator overloading

We discuss operator overloading in next chapter.

Function Overloading

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

The following example shows using function **print()** to print different data types:

```
using System;
namespace PolymorphismApplication
{
   class Printdata
   {
      void print(int i)
      {
         Console.WriteLine("Printing int: {0}", i );
      }

      void print(double f)
      {
         Console.WriteLine("Printing float: {0}" , f);
      }

      void print(string s)
      {
         Console.WriteLine("Printing string: {0}", s);
```

```
        }
        static void Main(string[] args)
        {
            Printdata p = new Printdata();

            // Call print to print integer
            p.print(5);

            // Call print to print float
            p.print(500.263);

            // Call print to print string
            p.print("Hello C++");
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Printing int: 5
Printing float: 500.263
Printing string: Hello C++
```

Dynamic Polymorphism

C# allows you to create abstract classes that are used to provide partial class implementation of an interface. Implementation is completed when a derived class inherits from it. **Abstract** classes contain abstract methods, which are implemented by the derived class. The derived classes have more specialized functionality.

Here are the rules about abstract classes:

- You cannot create an instance of an abstract class
- You cannot declare an abstract method outside an abstract class
- When a class is declared **sealed**, it cannot be inherited, abstract classes cannot be declared sealed.

The following program demonstrates an abstract class:

```
using System;
```

```
namespace PolymorphismApplication
{
   abstract class Shape
   {
      public abstract int area();
   }
   class Rectangle:  Shape
   {
      private int length;
      private int width;
      public Rectangle( int a=0, int b=0)
      {
         length = a;
         width = b;
      }
      public override int area ()
      {
         Console.WriteLine("Rectangle class area :");
         return (width * length);
      }
```

```
    }

    class RectangleTester
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle(10, 7);
            double a = r.area();
            Console.WriteLine("Area: {0}",a);
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Rectangle class area :
Area: 70
```

When you have a function defined in a class that you want to be implemented in an inherited class(es), you use **virtual** functions. The

virtual functions could be implemented differently in different inherited class and the call to these functions will be decided at runtime.

Dynamic polymorphism is implemented by **abstract classes** and **virtual functions**.

The following program demonstrates this:

```
using System;
namespace PolymorphismApplication
{
   class Shape
   {
      protected int width, height;
      public Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      public virtual int area()
      {
         Console.WriteLine("Parent class area :");
```

```
        return 0;
    }
}
class Rectangle: Shape
{
    public Rectangle( int a=0, int b=0): base(a, b)
    {

    }
    public override int area ()
    {
        Console.WriteLine("Rectangle class area :");
        return (width * height);
    }
}
class Triangle: Shape
{
    public Triangle(int a = 0, int b = 0): base(a, b)
    {
```

```
    }
    public override int area()
    {
        Console.WriteLine("Triangle class area :");
        return (width * height / 2);
    }
}
class Caller
{
    public void CallArea(Shape sh)
    {
        int a;
        a = sh.area();
        Console.WriteLine("Area: {0}", a);
    }
}
class Tester
{

    static void Main(string[] args)
```

```
        {
            Caller c = new Caller();
            Rectangle r = new Rectangle(10, 7);
            Triangle t = new Triangle(10, 5);
            c.CallArea(r);
            c.CallArea(t);
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Rectangle class area:
Area: 70
Triangle class area:
Area: 25
```

C# - Interfaces

An interface is defined as a syntactical contract that all the classes inheriting the interface should follow. The interface defines the **'what'** part of the syntactical contract and the deriving classes define the **'how'** part of the syntactical contract.

Interfaces define properties, methods, and events, which are the members of the interface. Interfaces contain only the declaration of the members. It is the responsibility of the deriving class to define the members. It often helps in providing a standard structure that the deriving classes would follow.

Abstract classes to some extent serve the same purpose, however, they are mostly used when only few methods are to be declared by the base class and the deriving class implements the functionalities.

Declaring Interfaces

Interfaces are declared using the interface keyword. It is similar to class declaration. Interface statements are public by default. Following is an example of an interface declaration:

```
public interface ITransactions
{
```

```
   // interface members
   void showTransaction();
   double getAmount();
}
```

Example

The following example demonstrates implementation of the above interface:

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

namespace InterfaceApplication
{
   public interface ITransactions
   {
      // interface members
      void showTransaction();
      double getAmount();
   }
```

```
public class Transaction : ITransactions
{
    private string tCode;
    private string date;
    private double amount;
    public Transaction()
    {
        tCode = " ";
        date = " ";
        amount = 0.0;
    }

    public Transaction(string c, string d, double a)
    {
        tCode = c;
        date = d;
        amount = a;
    }
```

```
        public double getAmount()
        {
            return amount;
        }

        public void showTransaction()
        {
            Console.WriteLine("Transaction: {0}", tCode);
            Console.WriteLine("Date: {0}", date);
            Console.WriteLine("Amount: {0}", getAmount());
        }
    }
    class Tester
    {
        static void Main(string[] args)
        {
            Transaction t1 = new Transaction("001",
"8/10/2012", 78900.00);
            Transaction t2 = new Transaction("002",
"9/10/2012", 451900.00);
            t1.showTransaction();
```

```
            t2.showTransaction();
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Transaction: 001
Date: 8/10/2012
Amount: 78900
Transaction: 002
Date: 9/10/2012
Amount: 451900
```

Abstract classes

Abstract classes, marked by the keyword abstract in the class definition, are typically used to define a base class in the hierarchy. What's special about them, is that you can't create an instance of them - if you try, you will get a compile error. Instead, you have to subclass them, as taught in the chapter on inheritance, and create an instance of your subclass. So when do you need an abstract class? It really depends on what you do.

To be honest, you can go a long way without needing an abstract class, but they are great for specific things, like frameworks, which is why you will find quite a bit of abstract classes within the .NET framework it self. A good rule of thumb is that the name actually makes really good sense - abstract classes are very often, if not always, used to describe something abstract, something that is more of a concept than a real thing.

In this example, we will create a base class for four legged animals and then create a Dog class, which inherits from it, like this:

```
namespace AbstractClasses
{
    class Program
    {
        static void Main(string[] args)
        {
            Dog dog = new Dog();
```

```
            Console.WriteLine(dog.Describe());
            Console.ReadKey();
        }
    }

    abstract class FourLeggedAnimal
    {
        public virtual string Describe()
        {
            return "Not much is known about this
four legged animal!";
        }
    }

    class Dog : FourLeggedAnimal
    {

    }
}
```

If you compare it with the examples in the chapter about inheritance, you won't see a big difference. In fact, the abstract keyword in front of the FourLeggedAnimal definition is the biggest difference. As you can see, we create a new instance of the Dog class and then call the inherited Describe() method from the FourLeggedAnimal class. Now try creating an instance of the FourLeggedAnimal class instead:

```
FourLeggedAnimal someAnimal = new
FourLeggedAnimal();
```

You will get this fine compiler error:

Cannot create an instance of the abstract class or interface 'AbstractClasses.FourLeggedAnimal'

Now, as you can see, we just inherited the Describe() method, but it isn't very useful in it's current form, for our Dog class. Let's override it:

```
class Dog : FourLeggedAnimal
{
    public override string Describe()
    {
        return "This four legged animal is a
Dog!";
    }
}
```

In this case, we do a complete override, but in some cases, you might want to use the behavior from the base class in addition to new functionality. This can be done by using the base keyword, which refers to the class we inherit from:

```
abstract class FourLeggedAnimal
{
    public virtual string Describe()
    {
        return "This animal has four legs.";
```

```
    }
}


class Dog : FourLeggedAnimal
{
    public override string Describe()
    {
        string result = base.Describe();
        result += " In fact, it's a dog!";
        return result;
    }
}
```

Ref :

http://www.tutorialspoint.com/

http://www.c-sharpcorner.com/UploadFile/tusharkantagarwal/objectorientedcsharp11162005070743AM/objectorientedcsharp.aspx