Unit 1.3
Language Preliminaries & .Net Basic (8-Hrs)

Class and Object

- A class is a user-defined blueprint or prototype from which objects are created.
- Basically, a class combines the fields and methods (member function which defines actions) into a single unit.
- A programmer can create any number of objects of the same class.
- Classes have the property of information hiding. It allows data and functions to be hidden, if necessary, from external use.
- Since class is a collection of logically related data items and the associated functions which operate and manipulate those data. This entire collection can be called a new user defined data-type. So, classes are user-defined data types and behave like the built-in types of a programming language.
- Once a class has been specified (or declared), we can create variables of that type (by using the class name as datatype).

Declaration of Class

Syntax:

```
Access_Specifier class class_name
{
    //Field Variable
    //Mthods
}
```

Access Specifier can be public or internal. By default, it is **internal**. The internal access specifier hides its member variables and methods from other classes and objects, that is resides in other namespace. The variable or classes that are declared with internal can be access by any member within application. It is the default access specifiers for a class in C# programming.

Creating Object

- The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory.
- The new operator also invokes the class constructor.

  Syntax:

  Class_Name obj=new Class_Name(Type arg1, Type arg2….)

Example:

- **Create a class Rectangle with two function SetData(double, double) for setting the value of length and breadth and double GetArea() to return the area of rectangle.**

```
using System;
namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Enter length and breadth  of rectangel : ");
            double length = Convert.ToDouble(Console.ReadLine());
            double breadth = Convert.ToDouble(Console.ReadLine());
            Rectangle r = new Rectangle();
            r.SetData(length, breadth);
            double res = r.GetArea();
            Console.WriteLine("Area : "+res);
            Console.ReadKey();
        }
    }
    class Rectangle
    {
        double length, breadth;
        public void SetData(double length, double breadth)
        {
            this.length = length;
            this.breadth = breadth;
        }
```

```
        public double GetArea()
        {
            return (length * breadth);
        }
    }
}
```

```
Enter length and breadth  of rectangel :
5
6
Area : 30
```

Constructor and its Types

- A constructor is a special **member function** having the same name as that of the class which is used to automatically initialize the objects of the class type with legal initial values.
- The constructor is invoked automatically whenever an object of its associated class is created using new keyword.
- It is called constructor because it constructs the values of data members of the class.
- Constructor are categorized into types: Default constructor and Parameterized Constructor.
- A constructor that does not take any parameter is called **default constructor.**
- The constructors that can take arguments or parameters are called **parameterized constructor.** In this, we pass the initial value as arguments to the constructor function when the object is declared.
- When more than one constructor functions are defined in the same class then we say the **constructor is overloaded i.e. constructor overloading**. All the constructors have the same name as the corresponding class, and they differ in terms of number of arguments, data types of argument or both. This makes the creation of object flexible.

A constructor is declared and defined as below

```
class Demo
{
        public Demo(); //Default constructor
        {
                //body of constructor
        }
        public Demo(int a ,int b ) //Parameterized Constructor
        {

        }
}
```

- **Demonstrate the concept of constructor to find the area of rectangle.**

```
using System;
namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Enter Length and Breadth of Rectangle");
            double len = Convert.ToDouble(Console.ReadLine());
            double bre = Convert.ToDouble(Console.ReadLine());
            Rectangle r = new Rectangle(len, bre);
            double res = r.GetArea();
            Console.WriteLine("Area of Rectangle : " + res);
            Console.ReadKey();
        }


        class Rectangle
        {
            double length, breadth;
```

```
            public Rectangle(double length, double breadth)
            {
                this.length = length;
                this.breadth = breadth;
            }
            public double GetArea()
            {
                return (length * breadth);
            }
        }
    }
}
```

- **Create a class Person with data member name, age, address and citizenship_number. Write a constructor to initialize the value of a person. Assign citizenship number if the age of the person is greater than 18 otherwise assign values zero to citizenship number. Also, create a function to display the values. Write a main program to test your class.**

```
using System;
namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Enter Name of a person");
            string name = Console.ReadLine();
            Console.Write("Enter Address");
            string address = Console.ReadLine();
            Console.WriteLine("Enter Age");
            int age = Convert.ToInt32(Console.ReadLine());
            Int64 cn = 0;
            if(age>18)
            {
                Console.Write("Enter citizenship number");
                cn = Convert.ToInt64(Console.ReadLine());
            }
            Person p = new Person(name, address, age, cn);
            p.DisplayDetail();
            Console.ReadKey();
        }
    }
    class Person
    {
        String Name, Address;
        int age;
        Int64 citizenshipnumber;
        public Person(String Name, String Address, int age, Int64 citizenshipnumber)
        {
            this.Name = Name;
            this.age = age;
            this.Address = Address;
            this.citizenshipnumber = citizenshipnumber;
        }
        public void DisplayDetail()
        {
            Console.WriteLine("Name =" + Name);
            Console.WriteLine("Address=" + Address);
            Console.WriteLine("Age= " + age);
            Console.WriteLine("Citizenship Number=" + citizenshipnumber);
        }
    }
}
```

Structure

- Structures provide a method for packing together data of different types.
- A structure is a convenient tool for handling a group of logically related data items. Once structure has been defined, we can create variables of that type using declaration that are similar to built in data type declarations.
- It helps you to make a single variable hold related data of various data types.
- The struct keyword is used for creating a structure.
- Structures are used to represent a record.
- Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book
    - Title
    - Author
    - Subject
    - Book ID

Syntax:

```
struct  <Struct_Name>
{
        // Structure Constructor, Data Members, Function Members
};
```

Example:

```
struct Books
{
        public string title;       //if not specified, the access specifier is private by default
        public string author;
        public string subject;
        public int book_id;
};

or
struct Books
{
        private string title;
        private string author;
        private string subject;
        private int book_id;
        public Books()
        {
                //Constructor Defination
        }
         public void SetData(String title, String author, String subject, int book_id)
        {
                //Function Defination
        }
       public  void DisplayData()
       {
                //Function Defination
       }
};
```

Structure vs Class

1. Structure is declared with struct keyword while class is declared with class keyword.
2. Structure are only object based but class support all features of OOP.
3. Structure doesn't support inheritance but class supports inheritance.
4. In structure data member can't be initialized in structure definition but in class data member can be initialized in class definition.
5. Structure members cannot be specified as abstract or protected.

6. structs can be instantiated with or without using the New operator but class must be instantiated with new keyword.
7. Class is reference type but structure is value type.

Note:

- *There are two kinds of types in C#: reference types and value types. Variables of reference types store references to their data (objects), while variables of value types directly contain their data. With reference types, two variables can reference the same object; therefore, operations on one variable can affect the object referenced by the other variable. With value types, each variable has its own copy of the data, and it is not possible for operations on one variable to affect the other.*

- The following program demonstrates the concept of c# structure.

```
using System;
namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Books b = new Books();
            b.SetData("Netcentric Computing", "Bhesh", "Programming", 1);
            b.DisplayData();
            Console.ReadKey();
        }
    }

    struct Books
    {
        private string title;
        private string author;
        private string subject;
        private int book_id;
        public void SetData(String title, String author, String subject, int book_id)
        {
            this.title = title;
            this.author = author;
            this.subject = subject;
            this.book_id = book_id;
        }
        public void DisplayData()
        {
            Console.WriteLine("Book Title : " + title);
            Console.WriteLine("Book Author : " + author);
            Console.WriteLine("Subject : " + subject);
            Console.WriteLine("Book ID : " + book_id);
        }
    };
}
```

```
Book Title : Netcentric Computing
Book Author : Bhesh
Subject : Programming
Book ID : 1
```

Properties

- Properties are special type of methods that execute when you **get (Read) or set (Write)** data. The data is commonly stored in a field, but could be stored externally, or calculated at runtime.
- The get accessor return property value while set accessor initialize property value.
- Properties are the preferred way to encapsulate fields unless the memory address of the field needs to be exposed.
- A property is simply **a method that acts and looks like a field** when you want to get or set a value, thereby **simplifying the syntax.** For, example: It removes manual definition of a method named like GetName() that returned a string containing the name and SetName(String) that assigns string value to a string field.
- Types:

1. Read-only(GET) Property: A read only property only has a get implementation.
2. Settable(SET) Property: A settable property has both get and set implementation.

- The following program demonstrates using property.

WAP to input name and roll of students. Use the concept of property to get and set the data.

```
using System;
namespace ConsoleApp
{
    class Program
    {

        static void Main(string[] args)
        {
            Student s = new Student();
            s.Name = "Ram";
            s.Roll = 5;
            Console.WriteLine("Name=" + s.Name);
            Console.WriteLine("Roll=" + s.Roll);
            Console.ReadKey();
        }
    }
    class Student
    {
         public String Name { get; set; }   //Property Name
         public int Roll { get; set; } //Property Roll
    }
}
```

Note: Properties can also be defined as below

```
class Student
{
        string name; //Property
        public string Name
        {
            Get           //Get The Value of Property
            {
                return (name);
            }
            set          //Set the Value of Property
            {
                name = value;   //Value is implicit keyword
            }
        }
}
```

Indexers

- Indexers are also special methods that execute when we get or set data using array syntax [].
- Indexers allow the calling code to use the array syntax to access a property.
- **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 ([ ]).
- Indexers allow instances of a class or struct to be indexed just like arrays. The indexed value can be set or retrieved without explicitly specifying a type or instance member.

Syntax:

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

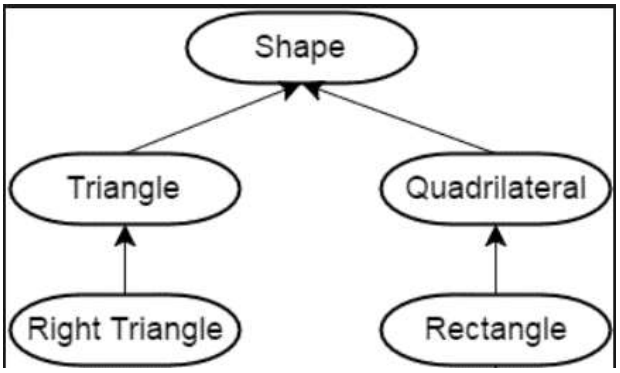
```
        // set the value specified by index
    }
}
```

- 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. Indexers resemble properties except that their accessors take parameters.
- 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.
- Properties doesn't have parameter but indexers are properties with parameter.
- Indexers are accessed using indexes but properties are accessed by names.
- Indexers is an instance member so can't be static but property can be static.
- The following example demonstrates the concept.

```
using System;
namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Student_Name s = new Student_Name();
            Console.WriteLine("Enter name of 5 students");
            for(int i=0;i<5;i++)
            {
                s[i] = Console.ReadLine();
            }
            Console.WriteLine("Name of Students are ");
            for(int i=0;i<5;i++)
            {
                Console.WriteLine(s[i]);
            }
            Console.ReadKey();
        }
    }
    class Student_Name
    {
        String  [] Name = new string[5];
        public String this [int index]
        {
            get
            {
                return (Name[index]);
            }
            set
            {
                Name[index] = value;
            }
        }
    }
}
```

Inheritance

- Inheritance is the mechanism of deriving a new class from existing class.
- The existing class is called super class or base class or parent class and the child class is called subclass or derived class or extended class.
- The subclass inherits some of the properties from the base class and can add or extends its own property as well.
- All the public and protected properties such as fields, methods etc. can be inherited by the derived class however private members can't be inherited.
- In c#, we can achieve inheritance using ":" operator.



- The main advantage of inheritance is the concept of **code reusability.** A code is said to be reusable if we can reuse the properties of superclass in other class by using the concept of inheritance. A programmer can use a class created by another person or company without any modification at all or with small modification such as deriving other classes from it to fit his situation. So reusing existing codes saves time and money and increase code reliability.

Inheritance Types

1. Single inheritance
2. Multiple inheritance
3. Hierarchical inheritance
4. Multilevel inheritance
5. Hybrid inheritance

1. Single inheritance

In single inheritance, a class is derived from only one existing class. i.e. only one super class.
The general form is given below

```
class A
{
        //member of A
}
class B : A
{
        //own member of B
}
```

2. Multiple inheritance

In multiple inheritance, a class is derived from more than one existing classes. **C# doesn't support multiple inheritance due to ambiguity in parent class**. When compilers of programming languages that support this type of multiple inheritance encounter super classes that **contain methods with the same name,** they sometimes cannot determine which member or method to access or invoke. We use alternative approach '**interface**' to implement the concept of multiple inheritance in java without any ambiguity.

3. Hierarchical inheritance

In hierarchical inheritance, two or more classes inherits the properties of one existing class. The general form is

```
class A
{
        //member of A
}

class B : A
{
        //own member of B
}
```

```
class C : A
{
        //own member of  C
}
```

4. Multilevel inheritance
   In multilevel inheritance, a class is derived from another subclass. The general form is

```
class A
{
        //member of A
}
class B : A
{
        //own member of B
}
class C : B
{
        //own member of C
}
```

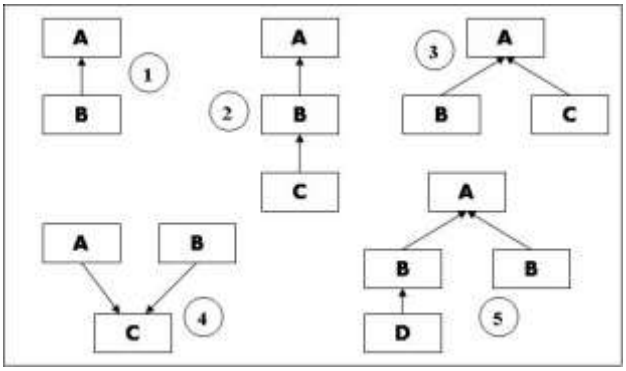
5. Hybrid inheritance
   In hybrid inheritance, it can be the combination of single and multiple inheritance or any other combination. One form can be as below

```
Class X
{
        //member of X
}
Class A : X
{
        // own member of A
}
Class B : A
{
        //own member of C
}
Class C : A
{
        //own member of C
}
Class D : A
{
        //own member of D
}
```

Above example consists both single and hierarchical inheritance hence called hybrid inheritance.

The below figure illustrates inheritance.

- (1) Simple / Single
- (2) Multilevel
- (3) Hierarchical
- (4) Multiple
- (5) Hybrid



A class Room consists of two fields length and breadth and method int Area () to find the area of room. A new class BedRoom is derived from class Room and consist of additional field height and two method SetData(int,int,int) to set the value for three fields and int Volume() to find the volume. Now write the main program to input the length, breadth and height and find the area and volume.

```
using System;
namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Enter length, breadth and height");
            int l = Convert.ToInt16(Console.ReadLine());
            int b = Convert.ToInt16(Console.ReadLine());
            int h = Convert.ToInt16(Console.ReadLine());
            BedRoom br = new BedRoom();
            br.SetData(l, b, h);
            int area = br.Area();
            int vol = br.Volume();
            Console.WriteLine("Area : " + area);
            Console.WriteLine("Volume : " + vol);
            Console.ReadKey();
        }
    }
    class Room
    {
        protected int length, breadth;
        public int Area()
        {
            return (length * breadth);
        }
    }
    class BedRoom : Room  //BedRoom Now Inherits characteristics of Class Room
    {
        private int height;
        public void SetData(int length, int breadth, int height)
        {
            this.length = length;
            this.breadth = breadth;
            this.height = height;
        }
        public int Volume()
        {
            return (Area() * height);
        }
    }
}
```

Method Hiding

- C# provides a concept to hide the methods of the base class from derived class, this concept is known as Method Hiding. It is also known as Method Shadowing.
- In method hiding, we can hide the implementation of the methods of a base class from the derived class using the **new** keyword. Or in other words, in method hiding, you can **redefine** the method of the base class in the derived class by using the **new keyword**.
- The following program demonstrates the concept

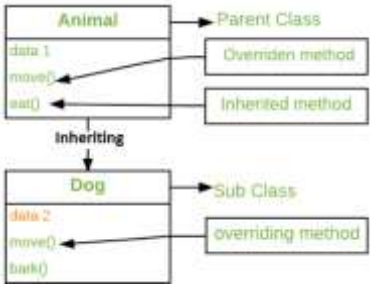
```
using System;
namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Demo2 d = new Demo2();
            d.Display();
            Console.ReadKey();
        }
    }
    class Demo1
    {
        public void Display()
        {
            Console.Write("Base Class Method Invoked");
        }
    }
    class Demo2: Demo1
    {
        public new void Display()
        {
            Console.Write("Derived Class Method Invoked");
        }
    }
}
```

```
Derived Class Method Invoked
```

- So in above program, We have defined two methods with same name i.e. Display() in base class as well as derived class. So, when we redefine the same function with new keyword in derived class then derived class method will hide the base class method. This is useful when it is required to change the behavior of derived class redefined method.

Method Overriding

- The process of redefining the inherited methods of the base class in the derived class is called method overriding.



- The method in the derived class should have same signature and same return type as that of base class in order to override it.
- Method Overriding is a technique that allows the invoking of functions from another class (base class) in the derived class.
- In simple words, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature and same return type (or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class. Method overriding is one of the ways by which C# achieve **Run Time Polymorphism (Dynamic Polymorphism or Dynamic Method Dispatch).**

- Runtime Polymorphism is about allowing a derived class to override an inherited action to provide custom behaviour.
- **Polymorphism promotes extensibility** i.e. we can introduce other specific type implementations in our inheritance hierarchy and the only required changes to incorporate them in our system are in the extended code i.e. derived unit, no changes are required to our base unit. Our base unit which invokes the polymorphic method is independent of the object types considering of course the fact that all these types are in the same inheritance hierarchy.
- The method that is overridden by an override declaration is called the **overridden base method**.
- An override method is a **new implementation** of a member that is inherited from a base class.
- Rather than hiding a method, it is usually better to override it. We can only override if the base class chooses to allow overriding, by applying the virtual keyword
- We use reference to the base class to refer to all derived objects. But the problem here is that, it always executes the function in the base class as demonstrated by the below program.

```
using System;
namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Demo1 d; //d1 is reference type
            Demo1 d1 = new Demo1();
            d = d1; //Access Base Class Method via Base Class Reference Type
            d1.Display();
            Demo2 d2= new Demo2();
            d = d2;
            d.Display();//Again Access Base Class Method via Base Class Reference Type
            Console.ReadKey();
        }
    }
    class Demo1
    {
        public void Display()
        {
            Console.WriteLine("Base Class Method Invoked");
        }
    }
    class Demo2: Demo1
    {
        public void Display()
        {
            Console.WriteLine("Derived Class Method Invoked");
        }
    }
}
```

```
Base Class Method Invoked
Base Class Method Invoked
```

- To select the appropriate member function while the program is running i.e. run time polymorphism, c# provide a mechanism that make use of **virtual and override keyword.**
- When same function name is used in base and derived classes, the function in base class is declared as virtual using keyword virtual preceding its normal declaration. And similarly, the derived class function is declared as overridden using override keyword preceding its normal declaration.
- Hence we can execute different versions of overridden functions by making base Reference type Reference to different derived class object to achieve run time polymorphism or dynamic polymorphism.

```
using System;
namespace ConsoleApp
{
    class Program
```

```
    {
        static void Main(string[] args)
        {
            Demo1 d; //d1 is reference type
            Demo1 d1 = new Demo1();
            d = d1; //Access Base Class Method via Base Class Reference Type
            d1.Display();
            Demo2 d2= new Demo2();
            d = d2;
            d.Display();//Again Access Base Class Method via Base Class Reference Type
            Console.ReadKey();
        }
    }
    class Demo1
    {
        public virtual void Display()
        {
            Console.WriteLine("Base Class Method Invoked");
        }
    }
    class Demo2: Demo1
    {
        public override void Display()
        {
            Console.WriteLine("Derived Class Method Invoked");
        }
    }
}
```

```
Base Class Method Invoked
Derived Class Method Invoked
```

Abstract Class

- Abstraction in C# is the process to hide the internal details and showing only the functionality.
- A class is said to be abstract base class or simply abstract class, if we can't instantiate object of that class without creating its sub class.
- Such class only exists as parent of derived classes from which objects are instantiated.
- Abstract Class is a class which contains at least one abstract method.
- Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the abstract method.
- If we need our method to be always overridden, then we can declare that method as abstract method using abstract modifier and without method definition. Thus declared methods of an abstract class must be defined in its subclass i.e. concrete class.
- A derive class that implements all the missing functionality is called concrete class. So, the concrete class provides implementations for the member functions that are not implemented in the base class.
- Concrete class is a complete class but abstract class is incomplete superclass.
- Abstract class can't be instantiated but concrete class can be instantiated.
- The following program demonstrates the concept.

```
using System;
namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Demo d; //D is Reference Type variable of Abstract Class Demo
            Demo1 d2 = new Demo1();
            d = d2;
            d.Display();
            Console.ReadKey();
        }
    }
```

```
    abstract class Demo  //Abstract Class
    {
       public abstract void Display(); //Abstract Method
    }

     class Demo1 : Demo //Concrete Class
     {
         public override void Display()
         {
             Console.Write("Derived Class Method Invoked");
         }
     }
}
```

```
Derived Class Method Invoked
```

Use of Base Keyboard

- The base keyword allows a subclass to access members of its super class i.e. the base class that it inherits or derives from.
- It basically used to access **constructors and methods** or functions of the base class.
- Base keyword specifies which constructor of the base class should be invoked while creating the instances of the derived class.
- Uses
  1. Call methods or functions of base class from derived class.
  2. Call constructor internally of base class at the time of inheritance.

WAP to demonstrate the use of base keyword to call methods of base class from derived class.

```
using System;
namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Demo2 d = new Demo2();
            d.Display();
            Console.ReadKey();
        }
    }

    class Demo1
    {
        public virtual void Display()
        {
            Console.WriteLine("Base Class Method Invoked");
        }
    }
    class Demo2: Demo1
    {
        public override void Display()
        {
            base.Display();  //This will invoke Display() Method from Base Class
            Console.WriteLine("Derived Class Method Invoked");
        }
    }
}
```

```
Base Class Method Invoked
Derived Class Method Invoked
```

WAP to demonstrate the use of base keyword to call constructor of base class from derived class at the time of inheritance.

OR

A class Figure consist of two fields dim1 and dim2, constructor to construct these fields and an abstract method float area() to compute area. Define two classes Rectangle and Triangle both of which consists of overridden method float area() method and constructor that invoke Figure constructor. Now write a program to find the area of rectangle and triangle inside it.

```
using System;
namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Figure f;
            Triangle t = new Triangle(5, 6);
            f = t;
            double res=f.Area();
            Console.WriteLine("Area of Triangle : " + res);

            Rectange r = new Rectange(5,6);
            f = r;
            res = f.Area();
            Console.WriteLine("Area of Rectangle : " + res);
            Console.ReadKey();
        }
    }
    abstract class Figure
    {
        protected double dim1, dim2;
        public Figure(double dim1, double dim2)
        {
            this.dim1 = dim1;
            this.dim2 = dim2;
        }
        public abstract double Area();
    }
    class Rectange : Figure
    {
        public Rectange(double dim1, double dim2):base(dim1,dim2) //This will immediately
                                                                  //invoke base class constructor
        {
        }
        public override double Area()
        {
            return (dim1 * dim2);
        }
    }
    class Triangle: Figure
    {
        public Triangle(int dim1, int dim2):base(dim1,dim2)
        {
        }
        public override double Area()
        {
            return (1.0 / 2 * dim1 * dim2);
        }
    }
}
```

```
Area of Triangle : 15
Area of Rectangle : 30
```

Preventing Inheritance and Overriding : Sealed class and Sealed method

- Sealed classes are used to restrict the users from inheriting the class.
- A class can be sealed by using the sealed keyword or modifier. This keyword tells the compiler that the class is sealed, and therefore, cannot be extended.

- No class can be derived from a sealed class.
- Similarly, a method can also be sealed, and in that case, the method cannot be overridden. However, a method can be sealed in the classes in which they have been inherited. If you want to declare a method as sealed, then it has to be declared as **virtual** in its base class.
- Sealed method is implemented so that no other class can overthrow it and implement its own method.

```
//Sealed Class Example
 class Demo
 {

 }
 sealed class Demo1:Demo
 {

 }
 class Demo2 : Demo1   //This Statement generates compile time error as it attempts to inherit
                         //Sealed Class
 {

 }

 //Sealed Method Example
    class Demo
    {
        public virtual void display()
        {

        }
    }
    class Demo1:Demo
    {
        sealed public override void display()
        {

        }
    }
    class Demo2 : Demo1
    {
        public override void display()  //This Overridden Definition will generate compile time
                                        //error as it attempt to override Sealed Method.
        {

        }
    }
```

Interface

- An interface is a named collection of methods declaration without implementations/definition.
- Interface is similar to class, but they lack instant variable and their method are declared without body.
- Interface define what a class must do but not how it does.
- To declare an interface, use *interface* keyword.
- It is used to provide total abstraction. That means all the members in the interface are declared with the empty body and are public and **abstract by default**.
- A class that implements interface must implement all the methods declared in the interface.

Syntax: Creating Interface

```
interface  <interface_name >
{
        //Methods Declaration
}
```

Syntax: Implementing Interface

class class_name :< interface_name>

Class vs Interface

- Class doesn't support multiple inheritance but interface supports all the types of inheritance.
- Class can contain both concrete as well as abstract method but interface only contains methods declaration i.e. method without definition.
- Class can have any access specifiers but interface access specifier must be public.
- Class can have field but interface can't contain field.
- Class can have constructor but interface can't have constructor.
- Class can have main() method but interface can't have main() method.
- Class methods can be static but interface method can't be static.
- Class can be instantiated to create object of class but interface can't be instantiated.

Advantage of Interface

- It is used to achieve loose coupling. If the only knowledge that class A has about class B, is what class B has exposed through its interface, then class A and class B are said to be loosely coupled.
- It is used to achieve total abstraction.
- To achieve multiple inheritance and abstraction.
- Interfaces add a plug and play like architecture into applications.

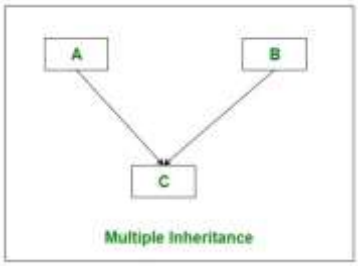
WAP to demonstrate the concept of interface.

```
using System;
namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Display d;
            d = new Monitor();
            d.ShowDisplay();
            d = new Printer();
            d.ShowDisplay();
            Console.ReadKey();
        }
    }
    interface Display //Interface Declaration
    {
        void  ShowDisplay();  //Methods Declaration
    }
    class Monitor : Display  //Implementing Interface
    {
        public void ShowDisplay()
        {
            Console.WriteLine("Displaying From Monitor");
        }
    }
    class Printer : Display
    {
        public void ShowDisplay()
        {
            Console.WriteLine("Display From Printer");
        }
    }
}
```

```
Displaying From Monitor
Display From Printer
```

Use of interface to achieve multiple inheritance

- In multiple inheritance, a class is derived from more than one existing classes. **C# doesn't support multiple inheritance due to ambiguity in parent class**. When compilers of programming languages that support this type of multiple inheritance encounter super classes that **contain methods with the same name,** they sometimes cannot determine which member or method to access or invoke. For example, in below figure if class A and B both have function named display(), the class C will get two versions of diplay() from its two parent class which creates ambiguity.
- We use alternative approach '**interface**' to implement the concept of multiple inheritance in java without any ambiguity.
- In c#, a class can inherit i.e. extends only one class **but interface can inherit i.e. extends more than one interface and also class can implement more than one interface.**
- So if the situation come where we need to gather the properties of two or more than two kind of objects then we can use interface two achieve the **multiple inheritance.**



- So, with the help of the interface, class C(as shown in the above diagram) can get the features of interface A and interface B.
- **The following program demonstrates the concept.**
  Program 1: Cass A and B have Different Functions

```
using System;
namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            C ob = new C();
            ob.DisplayA();
            ob.DisplayB();
            Console.ReadKey();
        }
    }
    interface A
    {
        void DisplayA();
    }
    interface B
    {
        void DisplayB();
    }
    class C : A, B //Multiple Inheritance
    {
        public void DisplayA()
        {
            Console.WriteLine("I am From Parent A");
        }
        public void DisplayB()
        {
            Console.WriteLine("I am From Parent B");
        }
    }
}
```

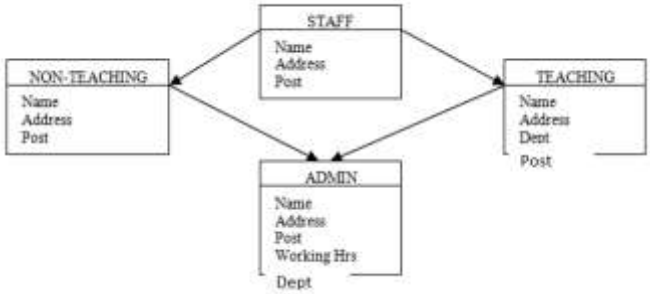
```
I am From Parent A
I am From Parent B
```

Program 2: Class A and B have same function

```
using System;
namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            A ob1 = new C();
            ob1.Display();
            B ob2 = new C();
            ob2.Display();
            Console.ReadKey();        }
    }
    interface A
    {
        void Display();
    }
    interface B
    {
        void Display();
    }
    class C : A, B //Multiple Inheritance
    {
        void A.Display()  //Resolving name ambiguity using interface name and . operator
        {
            Console.WriteLine("I am From Parent A");
        }
        void B.Display()  //Resolving name ambiguity using interface name and . operator
        {
            Console.WriteLine("I am From Parent B");
        }
    }
}
```

```
I am From Parent A
I am From Parent B
```

The following figure shows minimum information required for each class/interface. Specify all the classes/interfaces and define functions to assign and display individual information. Write a main program to test ADMIN, TEACHING and NON_TEACHING. What form of inheritance will the classes/interfaces hold in this case?




```
using System;
namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            ADMIN a = new ADMIN();
            a.SetName("Ram");
            a.SetAddress("Pokhara");
```

```
            a.SetPost("Administrator");
            a.setWorkingHour(5);
            a.setDepartment("IT");
            a.ShowName();
            a.ShowAddress();
            a.ShowPost();
            a.showWorkingHour();
            a.showDepartment();

            NON_TEACHING n = new ADMIN();
            n.SetName("Hari");
            n.SetAddress("Phoolbari");
            n.SetPost("Accountant");
            n.ShowName();
            n.ShowAddress();
            n.ShowPost();

            TEACHING t = new ADMIN();
            t.SetName("Shiva");
            t.SetAddress("Lamachaur");
            t.SetPost("Lecturer");
            t.setDepartment("IT");
            t.ShowName();
            t.ShowAddress();
            t.ShowPost();
            t.showDepartment();

            Console.ReadKey();
        }
    }

    interface STAFF
    {
        void SetName(string Name);
        void SetAddress(string Address);
        void SetPost(string Post);
        void ShowName();
        void ShowAddress();
        void ShowPost();
    }
    interface NON_TEACHING:STAFF
    {
    }



    interface TEACHING : STAFF
    {
        void setDepartment(string department);
        void showDepartment();
    }
    class ADMIN:NON_TEACHING,TEACHING
    {
        string Address,Department,Name,Post;
        int Working_Hour;
        public void SetAddress(string Address)
        {
            this.Address = Address;
        }
        public void setDepartment(string Department)
        {
            this.Department = Department;
        }
        public void SetName(string Name)
        {
            this.Name = Name;
        }
        public void SetPost(string Post)
        {
```

```
            this.Post = Post;
        }

        public void ShowAddress()
        {
            Console.WriteLine("Address : " + Address);
        }
        public void showDepartment()
        {
            Console.WriteLine("Department : " + Department);
        }
        public void ShowName()
        {
            Console.WriteLine("Name : " + Name);
        }
        public void ShowPost()
        {
            Console.WriteLine("Post : " + Post);
        }
        public void setWorkingHour(int Working_Hour)
        {
            this.Working_Hour = Working_Hour;
        }
        public void showWorkingHour()
        {
            Console.WriteLine("Working Hour : " + Working_Hour);
        }
    }
}
```

```
Name : Ram
Address : Pokhara
Post : Administrator
Working Hour : 5
Department : IT
Name : Hari
Address : Phoolbari
Post : Accountant
Name : Shiva
Address : Lamachaur
Post : Lecturer
Department : IT
```

Partial Class

- When working on large projects with multiple team members, it is useful to be able to split the definition of a complex class across multiple files.
- A partial class is a special feature of C#.
- It provides a special ability to implement the functionality of a single class into multiple files and all these files are combined into a single class file when the application is compiled.
- A partial class is created by using a *partial* keyword.

Advantage

1. When working on large projects, spreading a class over separate files enables multiple programmers to work on it at the same time.
2. We can also maintain your application in an efficient manner by compressing large classes into small ones.

Syntax:

```
public partial <class_name >
{
    // code
}
```

WAP to demonstrate the concept of partial class.

File 1: Program.cs

```
using System;
namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Student s = new Student("Ram",5);
            s.ShowDetail();
            Console.ReadKey();
        }
    }

}
```

File 2: Student1.cs

```
using System;
namespace ConsoleApp
{
        partial class Student
        {
            private string name;
            private int roll;
            public Student(string name, int roll)
            {
                this.name = name;
                this.roll = roll;
            }
        }
}
```

File 3: Student2.cs

```
using System;
namespace ConsoleApp
{
    partial class Student
    {
        public void ShowDetail()
        {
            Console.WriteLine("Name=" + name);
            Console.WriteLine("Roll=" + roll);
        }
    }
}
```

When we execute the above code, then compiler combines Student1.cs and Student2.cs into a single file, i.e. Student.

```
Name=Ram
Roll=5
```