(C# and .Net Framework)
Important questions

1

---

(1) Define Inheritance. What are the advantages of inheritance?

The technique of building a new class from an existing class is known as inheritance. Classes represent concepts and inheritance helps to represent hierarchical relationship between the concepts. Creating a new class from existing class is called as inheritance.

When a new class needs same members as an existing class then instead of creating those members again in new class, the new class can be created from existing class, which is called as inheritance. Main advantage of inheritance is **reusability** of the code.

(2) When to use inheritance?

1) Similarities : When similarities are existing between two or more classes, apply inheritance.
2) Extension : When additional features are needed in existing class, use inheritance and code only additional features in the new class.
3) Relationship : If relationship exist between classes such as IS-A, IS-A-KIND-OF, or IS-LIKE or IS-A-Type-OF, use inheritance
4) Generalization : Inheritance is advisible when generalization is fixed and does not require any modification or change.
5) Specialization : The inherited class is a specialized one having more data and functionalities.

(3) What is a derived class?

During inheritance, the class that is inherited is called as base class and the class that does the inheritance is called as derived class and every non private member in base class will become the member of derived class.

(4) Write syntax for creating a derived class in C#.

<u>Syntax</u>

```
[Access Modifier]  class  ClassName  :  base_classname

{

       -------
      -------

}
```

(5) Illustrate with an example inheritance in C#.

\

<u>Example implementing inheritance in C#.NET</u>

```
using System;

namespace ProgramCall
```

1

```
{
   class Base
   {
      int A, B;

      public void GetAB()
      {

         Console.Write("Enter Value For A : ");
         A = int.Parse(Console.ReadLine());
         Console.Write("Enter Value For B : ");
         B = int.Parse(Console.ReadLine());

      }

      public void PrintAB()
      {
         Console.Write("A  =  {0}\tB  =  {1}\t", A, B);

      }

   }
   class Derived : Base
   {
      int C;
      public void Get()
      {
         //Accessing base class method in derived class
         GetAB();

         Console.Write("Enter Value For C : ");
         C = int.Parse(Console.ReadLine());
      }

      public void Print()
      {

         //Accessing base class method in derived class
         PrintAB();

         Console.WriteLine("C  =  {0}", C);

      }
   }

   class MainClass
   {

      static void Main(string[] args)
      {
         Derived obj = new Derived();
         obj.Get();
         obj.Print();

         //Accessing base class method with derived class instance
         obj.PrintAB();

         Console.Read();
      }
   }
}
```

Output

```
Enter Value For A : 77
Enter Value For B : 88
Enter Value For C : 99
A  =  77        B  =  88        C  =  99
A  =  77        B  =  88
```

(6) What are different inheritance types?

Types of Inheritance in C#.NET

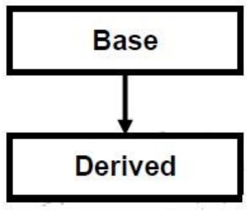
Creating a new class from existing class is called as inheritance.

Inheritance with Example

Inheritance can be classified to 5 types.

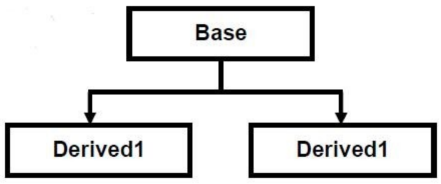
1. Single Inheritance
2. Hierarchical Inheritance
3. Multi Level Inheritance
4. Hybrid Inheritance
5. Multiple Inheritance
1. **Single Inheritance**

   when a single derived class is created from a single base class then the inheritance is called as single inheritance.



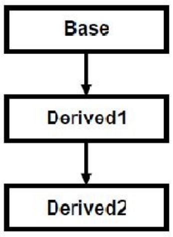
2. Hierarchical Inheritance

when more than one derived class are created from a single base class, then that inheritance is called as hierarchical inheritance.



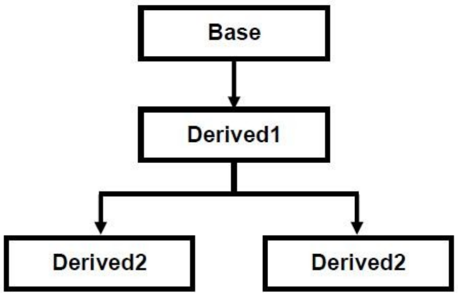
3. Multi Level Inheritance

when a derived class is created from another derived class, then that inheritance is called as multi level inheritance.



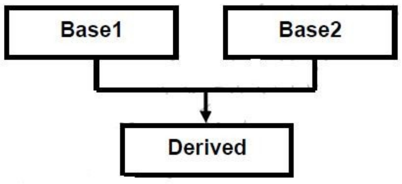
4. Hybrid Inheritance

Any combination of single, hierarchical and multi level inheritances is called as hybrid inheritance.



5. Multiple Inheritance

when a derived class is created from more than one base class then that inheritance is called as multiple inheritance. But multiple inheritance is not supported by .net using classes and can be done using interfaces.



Handling the complexity that causes due to multiple inheritance is very complex. Hence it was not supported in dotnet with class and it can be done with interfaces.

(7) What abstract classes? Illustrate with an example.

Abstract Classes and Abstract Methods in C#.NET

There may be a situation where it is not possible to define a method in base class and every derived class must override that method. In this situation abstract methods are used.

When a method is declared as abstract in the base class then it is not possible to define it in the derived class and every derived class of that class must provide its own definition for that method.

When a class contains at least one abstract method, then the class must be declared as **abstract class**.

It is **mandatory** to override abstract method in the derived class.

When a class is declared as abstract class, then it is not possible to create an instance for that class. But it can be used as a parameter in a method.

Example

The following example creates three classes shape, circle and rectangle where circle and rectangle are inherited from the class shape and overrides the methods Area() and Circumference() that are declared as abstract in Shape class and as Shape class contains abstract methods it is declared as abstract class.

```
using System;

namespace ProgramCall
{

   //Abstract class
   abstract class Shape1
   {

      protected float R, L, B;

      //Abstract methods can have only declarations
      public abstract float Area();
      public abstract float Circumference();

   }


   class Rectangle1 : Shape1
   {
      public void GetLB()
      {
         Console.Write("Enter  Length  :  ");

         L = float.Parse(Console.ReadLine());
```

```
            Console.Write("Enter Breadth : ");

            B = float.Parse(Console.ReadLine());
        }


        public override float Area()
        {
            return L * B;
        }

        public override float Circumference()
        {
            return 2 * (L + B);
        }

    }


    class Circle1 : Shape1
    {

        public void GetRadius()
        {

            Console.Write("Enter  Radius  :  ");
            R = float.Parse(Console.ReadLine());
        }

        public override float Area()
        {
            return 3.14F * R * R;
        }
        public override float Circumference()
        {
            return 2 * 3.14F * R;

        }
    }
    class MainClass
    {
        public static void Calculate(Shape1 S)
        {

            Console.WriteLine("Area : {0}", S.Area());
            Console.WriteLine("Circumference : {0}", S.Circumference());

        }
        static void Main()
        {
            Rectangle1 R = new Rectangle1();
            R.GetLB();
            Calculate(R);

            Console.WriteLine();

            Circle1 C = new Circle1();
            C.GetRadius();
            Calculate(C);

            Console.Read();

        }
    }
}
```

<u>Output</u>

```
Enter  Length  :  10
Enter Breadth : 12
Area : 120
Circumference : 44

Enter  Radius  :  5
Area : 78.5
Circumference : 31.4
```

<u>Note</u>

In the above example method calculate takes a parameter of type Shape1 from which rectangle1 and circle1 classes are inherited.

A base class type parameter can take derived class object as an argument. Hence the calculate method can take either rectangle1 or circle1 object as argument and the actual argument in the parameter S will be determined only at runtime and hence this example is an example for runtime polymorphism

(8) What are Sealed classes?

<u>Sealed classes and Sealed methods in C#.NET</u>

<u>Sealed Classes</u>

When you want to restrict your classes from being inherited by others you can create the class as sealed class.

To create a class as sealed class, create the class using the keyword sealed.

[Access Modifier] sealed class classname

{

}

<u>Sealed Methods</u>

The virtual nature of a method persists for any number of levels of inheritance.

For example there is a class “A” that contains a virtual method “M1”. Another class “B” is inherited from “A” and another class “C” is inherited from “B”. In this situation class “C” can override the method “M1” regardless of whether class “B” overrides the method “M1”.

At any level of inheritance, if you want to restrict next level of derived classes from overriding a virtual method, then create the method using the keyword **sealed** along with the keyword **override**.

[Access Modifier] sealed override returntype methodname([Params])

{

}

(9) What is Polymorphism? Illustrate polymosphism with an example.

Polymorphism (C# Programming Guide)
Visual Studio 2010

Polymorphism is often referred to as the third pillar of object-oriented programming, after encapsulation and inheritance. Polymorphism is a Greek word that means "many-shaped" and it has two distinct aspects:

1. At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays. When this occurs, the object's declared type is no longer identical to its run-time type.

2. Base classes may define and implement <u>virtual</u> methods, and derived classes can <u>override</u> them, which means they provide their own definition and implementation. At run-time, when client code calls the method, the CLR looks up the run-time type of the object, and invokes that override of the virtual method. Thus in your source code you can call a method on a base class, and cause a derived class's version of the method to be executed.

Virtual methods enable you to work with groups of related objects in a uniform way. For example, suppose you have a drawing application that enables a user to create various kinds of shapes on a drawing surface. You do not know at compile time which specific types of shapes the user will create. However, the application has to keep track of all the various types of shapes that are created, and it has to update them in response to user mouse actions. You can use polymorphism to solve this problem in two basic steps:

1. Create a class hierarchy in which each specific shape class derives from a common base class.
2. Use a virtual method to invoke the appropriate method on any derived class through a single call to the base class method.

First, create a base class called Shape, and derived classes such as Rectangle, Circle, and Triangle. Give the Shape class a virtual method called Draw, and override it in each derived class to draw the particular shape that the class represents. Create a List<Shape> object and add a Circle, Triangle and Rectangle to it. To update the drawing surface, use a foreach loop to iterate through the list and call the Draw method on each Shape object in the list. Even though each object in the list has a declared type of Shape, it is the run-time type (the overridden version of the method in each derived class) that will be invoked.

C#

```
public class Shape
{
   // A few example members
   public int X { get; private set; }
   public int Y { get; private set; }
   public int Height { get; set; }
   public int Width { get; set; }

   // Virtual method
   public virtual void Draw()
   {
      Console.WriteLine("Performing base class drawing tasks");
   }
}

class Circle : Shape
{
   public override void Draw()
   {
      // Code to draw a circle...
      Console.WriteLine("Drawing a circle");
      base.Draw();
   }
}
class Rectangle : Shape
{
   public override void Draw()
   {
      // Code to draw a rectangle...
      Console.WriteLine("Drawing a rectangle");
      base.Draw();
   }
}
class Triangle : Shape
{
   public override void Draw()
   {
      // Code to draw a triangle...
      Console.WriteLine("Drawing a triangle");
      base.Draw();
   }
}

class Program
{
    static void Main(string[] args)
    {
        // Polymorphism at work #1: a Rectangle, Triangle and Circle
        // can all be used whereever a Shape is expected. No cast is
        // required because an implicit conversion exists from a derived
        // class to its base class.
        System.Collections.Generic.List<Shape> shapes = new System.Collections.Generic.List<Shape>();
        shapes.Add(new Rectangle());
        shapes.Add(new Triangle());
        shapes.Add(new Circle());

        // Polymorphism at work #2: the virtual method Draw is
        // invoked on each of the derived classes, not the base class.
        foreach (Shape s in shapes)
        {
            s.Draw();
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

}

/* Output:
    Drawing a rectangle
    Performing base class drawing tasks
    Drawing a triangle
    Performing base class drawing tasks
    Drawing a circle
    Performing base class drawing tasks
 */
```

(10) What are interfaces? Illustrate with an example how interfaces are implemented in C#.

interface (C# Reference)
Visual Studio 2010

An interface contains only the signatures of methods, properties, events or indexers. A class or struct that implements the interface must implement the members of the interface that are specified in the interface definition. In the following example, class ImplementationClass must implement a method named SampleMethod that has no parameters and returns void.

For more information and examples, see Interfaces (C# Programming Guide).

Example

C#

```
interface ISampleInterface
{
   void SampleMethod();
}

class ImplementationClass : ISampleInterface
{
   // Explicit interface member implementation:
   void ISampleInterface.SampleMethod()
   {
      // Method implementation.
   }

   static void Main()
   {
      // Declare an interface instance.
      ISampleInterface obj = new ImplementationClass();

      // Call the member.
      obj.SampleMethod();
   }
}
```

An interface can be a member of a namespace or a class and can contain signatures of the following members:

- Methods
- Properties
- Indexers
- Events

An interface can inherit from one or more base interfaces.

When a base type list contains a base class and interfaces, the base class must come first in the list.

A class that implements an interface can explicitly implement members of that interface. An explicitly implemented member cannot be accessed through a class instance, but only through an instance of the interface.

For more details and code examples on explicit interface implementation, see Explicit Interface Implementation (C# Programming Guide).

The following example demonstrates interface implementation. In this example, the interface contains the property declaration and the class contains the implementation. Any instance of a class that implements IPoint has integer properties x and y.

C#

```
interface IPoint
{
  // Property signatures:
  int x
  {
    get;
    set;
  }

  int y
  {
    get;
    set;
  }
}

class Point : IPoint
{
  // Fields:
  private int _x;
  private int _y;

  // Constructor:
  public Point(int x, int y)
  {
    _x = x;
    _y = y;
  }

  // Property implementation:
  public int x
  {
    get
    {
      return _x;
    }

    set
    {
      _x = value;
    }
  }

  public int y
  {
    get
    {
      return _y;
    }
    set
    {
      _y = value;
    }
  }
}

class MainClass
{
  static void PrintPoint(IPoint p)
  {
    Console.WriteLine("x={0}, y={1}", p.x, p.y);
  }

  static void Main()
  {
    Point p = new Point(2, 3);
    Console.Write("My Point: ");
    PrintPoint(p);
  }
}
// Output: My Point: x=2, y=3
```

(11) Define structure b) Differentiate a structure from a class

Stucture is a composite data type in which the members of different data types are accomodated. It is a value type. It can be declared and created similar to a class type except the keyword struct is used in the place of class. A structure variable may be created without new operator.
A structure can have fields and methods.
It can have constructors ,but not Destructors.
The creation of structure results in creation on the
Stack and not on the heap.

SlNo	Struct	Class
1	Structure is a value type	A class is a reference type
2	Only constructor with parameters allowed	Constructors with or without parameters allowed
3	Destructor is not allowed	Destructor is allowed
4	No support for inheritance	Inheritance is supported
5	Passed by value	Passed by reference
6		

(12) Write a program to illustrate structure

```
using system;

class Student
{
struct Point(int a,int b)
{
x = a;
y = b;
}
public void readData()
{
Console.WriteLine(" Enter the values of X and Y : ");
x = convert.ToInt32(Connsole.ReadLine());
y = convert.ToInt32(Connsole.ReadLine());
}

public void writeData()
{
Console.WriteLine("The values of X and Y are : ");
Console.WriteLine("X = {0}",x:);
Console.WriteLine("Y = {0}",y);
}

public static void main()
{
Point p = new Point();
p.readData()
p.writeData();
Point p1 = new Point(10,20);
p1.writeData();
}
}
```

Program Output :

```
Enter the value of X and Y :
20
40
The x and y coordinates are a;
x = 20
y = 40
```

```
The x and y coordinates are a;
x = 10
y = 20
```

---

(13) Define a) Abstract classes b) Sealed classes

Abstract classes

In many applications ,we would like to have one base class and a number of different derived classes.
A Base Class simply act as a base for others ,we might not create its **objects**. Under such situations,a base class is decalred **abstract.**
The abstract modifier when used to declare a class cannot be instantiated. Only its classes can be instantiated. Only its derived classes can be instantiated.

Example

```
abstract class Base
{
 .......
}

class Derived : Base
{
.....
}
....
Base b1;   // Error

Derived d1;   // Correct
```

Sealed classes
  Some times,we may like to prevent a class being further subclassed for security reasons. A class that cannot be subclassed is called **Sealed Class.**

Example

```
Sealed class Aclass
{
...
}

Sealed class Bclass : Someclass
{
...
}
```

Any attempt to inherit sealed classes will cause an error and the compiler will not allow it.
Declaring a class **Sealed** prevent any unwanted extensions to the class.
A **Sealed** class cannot be an **abstract** class.

(14)   Differentiate class and an interface.

<u>Difference between a class and an Interface</u>

A **C# Class** is being considered the primary building block of the language. We use Classes as a template to put the properties and functionalities or behaviors in one building block for some group of objects and after that we use that template to create the objects we need.

A class can contain declarations of the following members:

Constructors, Destructors, Constants, Fields, Methods, Properties,Indexers, Operators, Events, Delegates, Classes, Interfaces, Structs

An **interface** contains only the signatures of methods, delegates or events. The implementation of the methods is done in the class that implements the interface. A class that implements an interface can explicitly implement members of that interface. An explicitly implemented member cannot be accessed through a class instance, but only through an instance of the interface.

```
interface ISampleInterface
{
    void SampleMethod();
}
class ImplementationClass : ISampleInterface
{
    // Explicit interface member implementation:
    void ISampleInterface.SampleMethod()
    {
        // Method implementation.
       Console.WriteLine(“ The Implementation class implements the SampleMethod
               specified in Interface IsampleInterface”);
    }
    static void Main()
    {
        // Declare an interface instance.
        ISampleInterface obj = new ImplementationClass();
        // Call the member.
        obj.SampleMethod();
    }
}
```

....

The Program Output

```
The Implementation class implements the SampleMethod

                  specified in Interface IsampleInterface
```

(15) Define a) Properties b) Indexer c) Event

Properties

One of the goals of OO Systems it not to permit any direct access to data members. It is normal practice to provide special methods known as accessor methods to have access to data members. Accessor methods can be used to set and and get the value of a private data member.
C# provides a mechanism known as **properties** that has the same capabilities as accesor methods,but it is much more elegant and simple to use. Using a property,a programmer can get access to data members as though they are public fields. Properties are some times refered to as **“smart fields”** as they add smartness to data fields.

Indexers

Indexers are location indicators and are used to access class objects,just like accessing elements in an array. An indexer allows accessing individual fields contained in the object of a class using subscript operator [] of an array.
Indexers are sometimes refered to as '**smart arrays**';
An indexer takes an *index* argument and looks like an array.
The indexer is declared using the keyword **this**.
The indexer is implemented through get and set accessors for the [] operator.

IMPLEMENTATION OF AN INDEXER

```
using system;
using System.Collections;
class List
{
  Arraylist array = new ArrayList();
public object this[int index]
{
get
{
If (index <0 || index >= array.Count)
{
return null;
}
else
{
return (array[index] = value;
}
}
}
class IndexTest
{
public static void Main()
{
List list = new List();
list[0] = “123”;

list[1] = “abc”;
list[2] = “xyz”;

for(int i =0; i<list.Count; i++)
{
Console.WriteLine(list[i]);
}
}
}
```

EVENT

An **event** is a delegate type class member that is used by the object or class to provide a notification to other objects that an event has occurred. The client object can act on an event by adding an event handler to the event.

Events are declared using the simple event declaration format as follows :

Modifier **event** type event name;

Examples

```
Public event EventHandler Click;
Public event RateChange Rate;
```

EventHandler and **RateChange** are delegates and **Click** and **Rate** are events;

IMPLEMENTING AN EVENT HANDLER

```
Using system;
//delegate declaration first

Public delegate void Edelegate(string str);

Class EventClass
{
//declaration of event
Public event Edelegate status;

Public void TriggerEvent()
{
If(status !- null}
Status(“Event Triggered);
}
}

Class EventTest
{
Public static void Main()
{
EventClass ec = new EventClass();
EventTest et = new EventTest();
Ec.Status += new EDelegate(et.Eventcatch);

Ec.TriggerEvent();

}

Public void EventCatch(string str)
{
Console.WriteLine(str);
}
}
```

(16) What are delegates? Explain the syntax for delegate declaration.

A Delegate in C# is a class type object and is used to invoke a method that has been encapsulated into it at the time of its creation.

Creating and using Delegates

Four steps

1) Delegate declaration
2) Delegate methods definition
3) Delegate instantiation
4) Delegate invocation

A Delegate declaration defines a class using the class **System.Delegate** as a base class.

Delegate methods are functions(defined in a class) whose signature matches the delegate signature exactly. The delegate instance holds the reference to delegate methods. The instance is used to invoke the methods indirectly.

DELEGATE DECLARATION

Modifier **delegate** return-type delegate-name(parameters);

DELEGATE METHODS

The methods whose references are encapsulated into a delegate instance are known as ***delegate methods*** or ***callable entities*****.** The signature and return type of delegate methods must exactly match the signature and return type of the delegate.

DELEGATE INSTANTIATION

The syntax for delegate instantiation is given below :
new delegate-type(expression);

DELEGATE INVOCATION

The syntax for delegate invocation :

delegate-object(parameter-list)**;**

CREATING AND IMPLEMENTING A DELEGATE

```
Using system;

//delegate declaration

delegate int ArithOp(int x,int y);

class MathOperation

{

//delegate methods definition

public static int Add(int a,int b)

{

return (a + b);

}

public static int Sub(int a,int b)

{

return (a - b);
}

}

class DelegateTest

{

public static void Main()

{

//delegate instances

ArithOp operation1 = new ArithOp(MathOperation.Add);

ArithOp operation2 = new ArithOp(MathOperation.Sub);

//Invoking Delegates
Int result1 = operation1(200,100);
Int result2 = operation2(200,100);
}
}
……
```

<u>Program Output</u>

```
Result1 = 300
Result2 = 100
```

(17) Explain briefly Publish/Subscribe Design Pattern.

Event

In a GUI environment, the occurrence of something significant is called an **event**.
A button-click, a menu selection etc are examples of events.

Publisher

A class containing visual components generating events in a Server Class, or server code is known as publisher.

Subscriber or Event Listners

A server class **publishes** an event that it can raise or generate.
Any number of classes can **listen or subscribe** to the published event. These classes are known as **client classes** or **subscribing classes** or client codes or **event listeners.**
A code containing a publishing class is known as **event source.**
When a publishing class of an event Source raises an event , all the subscribing classes are notified.
The subscribing classes implement the methods to handle the appropriate event.
The method that handles an event is known as an **event handler.**
This type of notifying and handling events is known as **publish/subscribe Design Pattern**

When an event is raised, the implementing methods(event handlers) in the subscribing classes are invoked through the delegate to handle the events.
The publish/subscribe design pattern is implemented by combining delegates and events.

(18) Define Operator Overloading.

Operator overloading is one of the exciting features of OOP. C# supports the idea of operator overloading. C# operators can be defined to work with the user defined data types such as structs and classes in much the same way as the built-in types. Operator overloading provides syntactic convenience by which c# operators can work with user defined data types.

OVERLOADABLE OPERATORS

Category	Operators
Binary Arithmetic	+,*,/,-,%
Unary arithmetic	+,-,++,--
Binary bitwise	&,\,^,<<,>>
Unary bitwise	!,~,true,false
Logical operators	==,!=,>=,<=,<,>
Conditional operators	&&,||
Compound assignment	+=,-+,*=,/=,%=
Other operators	[],(),=,?:,new,sizeof,typeof,is,as

NEED FOR OPERATOR OVERLOADING
Although operator overloading gives us syntactic convenience,it also help us greatly to generate more readable and intuitive code in a number of situations.

Example

- Mathematical or physical modelling where we use classes to represents objects such as coordinates,vectors,matrices,tensors,comples numbers and so on.

DEFINING OPERATOR OVERLOADING

To define an additional task to to an operator,we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special method called operator method,which describes the task.
The general form of Operator method is :

```
Public static retval operator op (arglist)
{
Method body  // task defined
}
```

EXAMPLE

```
Public static Complex Operator + (Complex c1,Complex C2)
{

Complex c3 = new Complex();
C3.x = c1.x + c2.x;
C3.y = c1.y + c2.y;
Return(c3)
}
```

(19) What is method overloading?

C# allows us to create more than one method with the same name,but with the different parameter lists and different definitions. This is called method overloading. Using the concept of method overloading,we can design a family of methods having the same name but using different argument lists.

EXAMPLE METHOD OVERLOADING

```
Using System;
Class Overloading
{
Public static void Main()
{
Console.WriteLine(Volume(10));
Console.WriteLine(Volume(2.5F,8));
Console.WriteLine(Volume(100L,75,15);
}
Static int Volume(int x)  // Cube
{
return (x * x * x);
}

Static double Volume(float r,int h)   // Cylinder
{
Return (3.14519 * r * r * h);
}

Static long Volume (long l,int b,int h)  // Box
{

Return ( l * b * h);
}
}

…………
```

Program Output

```
Volume of cube  = 1000
Volume of Cylinder = 157.2595
Volume of Box = 112500
```

(20) Write a program to declare a base class Printer. Derive classes from Printer to display details of different types of printers.

```
Using System;
Class Printer;
{
Public virtual void type()
{
Console.WriteLine("Printer Class");
}
Class InkJet : Printer
```

```
{
Public override void type()
{
Console.WriteLine(“Color Inkjet Printer”);
}

Class ElectroMechPrinter  : Printer
{
Public override void type()
{
Console.WriteLine(“ElectroMechanical Printer”);
}
}

Class DMP : Printer
{
Public override void type()
{
Console.WriteLine(“Dot Matrix Printer”);
}

}

Class  PrinterTest
{
Public static void Main()
{
Printer p = new printer();
ElectroMechPrinter e = new ElectroMechPrinter();
InkJet I = new InkJet();
DMP d = new DMP();
Printer p1 = I;

p.type();
e.type();
i.type();
d.type();
p1,type();

}
}
```

……………………
Program Output

```
Printer Class
Electro Mechanical Printer;
InkJet Printer;
DMP Printer;
Ink Jet Printer
```

(21) Write a program by having an abstract class Solid and implement Cylinder,Cone and Sphere by inheriting from Solid to find surface area and volume.

```
abstract class Shape
{
    public const double pi = Math.PI;
    protected double x, y;
    public Shape(double x, double y)
    {
        this.x = x;
        this.y = y;
    }
    public abstract double Area();
}
class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    {
    }
    public override double Area()
    {
        return pi * x * x;
    }
}
class Cylinder : Circle
{
    public Cylinder(double radius, double height)
        : base(radius)
    {
        y = height;
    }
    public override double Area()
    {
        return (2 * base.Area()) + (2 * pi * x * y);
    }
}
class TestShapes
{
    static void Main()
    {
        double radius = 2.5;
        double height = 3.0;
        Circle ring = new Circle(radius);
        Cylinder tube = new Cylinder(radius, height);
        Console.WriteLine("Area of the circle = {0:F2}", ring.Area());
        Console.WriteLine("Area of the cylinder = {0:F2}", tube.Area());
        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Area of the circle = 19.63
    Area of the cylinder = 86.39
*/
```

(22) Write a program to illustrate handling an event declared in an interface

```
Using system;
Public delegate void ButtonDelegate();                          // Delegate declaration
Public interface ButtonInterface
{
Event ButtonDelegate ButtonEvent;
Void OpenFile();
}
Public class ServerClass : ButtonInterface
{
Public event ButtonDelegate ButtonEvent;  //Event declaration
Public void OpenFile()     //method that calls the event
{
If (ButtonEvent != null )
ButtonEvent();  //Raising the event
}
}
Public class clientclass
{
Public clientclass
{
ButtonInterface Server = new ServerClass();

Server.ButtonEvent += new ButtonDelegate(file);       //Passing event handler
Server.OpenFile();
}

Private static void File() // Eventhandler
{
Console.WriteLine("A file is opened by clicking the button");
}
Public static void Main()
{
Cleintclass client = new clientclass();
}
}


………………………..
Program Output
A file is opened by clicking the button
```

(23) Write a program to illustrate overloading of binary operator + and *

```
public struct Complex
{
   public int real;
   public int imaginary;

   // Constructor.
   public Complex(int real, int imaginary)
   {
      this.real = real;
      this.imaginary = imaginary;
   }

   // Specify which operator to overload (+),
   // the types that can be added (two Complex objects),
   // and the return type (Complex).
   public static Complex operator +(Complex c1, Complex c2)
   {
      return new Complex(c1.real + c2.real, c1.imaginary + c2.imaginary);
   }

   // Override the ToString() method to display a complex number
   // in the traditional format:
   public override string ToString()
   {
      return (System.String.Format("{0} + {1}i", real, imaginary));
   }
}

class TestComplex
{
   static void Main()
   {
      Complex num1 = new Complex(2, 3);
      Complex num2 = new Complex(3, 4);

      // Add two Complex objects by using the overloaded + operator.
      Complex sum = num1 + num2;

      // Print the numbers and the sum by using the overridden
      // ToString method.
      System.Console.WriteLine("First complex number:  {0}", num1);
      System.Console.WriteLine("Second complex number: {0}", num2);
      System.Console.WriteLine("The sum of the two numbers: {0}", sum);

      // Keep the console window open in debug mode.
      System.Console.WriteLine("Press any key to exit.");
      System.Console.ReadKey();
   }
}
/* Output:
    First complex number:  2 + 3i
    Second complex number: 3 + 4i
    The sum of the two numbers: 5 + 7i
*/
```