# Objectives:

- **Abstract classes and Interface**
- **Properties and Indexers**

**1. Creating and using abstract class.**

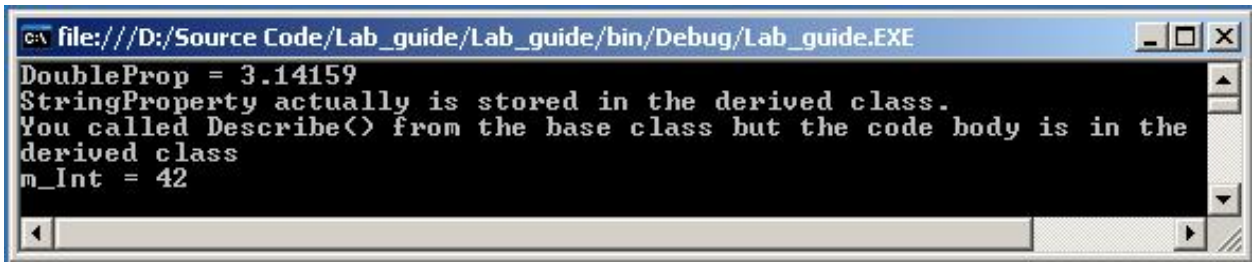
```
using System;
using System.Collections.Generic;
using System.Text;

namespace Lab_guide
{
    // Declare an abstract class

    abstract class clsBase
    {
        // Declare an abstract method. Note the semicolon to end the
declaration
        abstract public void Describe();

        // Declare an abstract property that has only a get accessor.
        // Note that you
        // do not prove the braces for the accessor
        abstract public double DoubleProp
        {
            get;
        }
        // Declare an abstract property that has only a set accessor.
        abstract public int IntProp
        {
            set;
        }
        // Declare an abstract propety that has both get and set
accessors. Note
        // that neither the get or set accessor may have a body.
        abstract public string StringProp
```

```
        {
            get;
            set;
        }
        // Declare a method that will access the abstract members.
        public void GetAbstract()
        {
            // Get the DoubleProp, which will be in the derived class.
            Console.WriteLine("DoubleProp = " + DoubleProp);
            // You can only set the IntProp value. The storage is in
the
            // derived class.
            IntProp = 42;

            // Set the StringProp value
            StringProp = "StringProperty actually is stored in " +
                         "the derived class.";
            // Now show StringProp
            Console.WriteLine(StringProp);

            // Finally, call the abstract method
            Describe();
        }
    }
    // Derive a class from clsBase. You must implement the abstract
members
    class clsDerived : clsBase
    {
        // Declare a constructor to set the DoubleProp member
        public clsDerived(double val)
        {
            m_Double = val;
        }
        // When you implement an abstract member in a derived class,
you may not
        // change the type or access level.
        override public void Describe()
```

```
        {
            Console.WriteLine("You called Describe() from the base " +
                              "class but the code body is in the \r\n"
+
                              "derived class");
            Console.WriteLine("m_Int = " + m_Int);
        }

        // Implement the DoubleProp property. This is where you provide
a body
        // for the accessors.
        override public double DoubleProp
        {
            get { return (m_Double); }
        }
        // Implement the set accessor for IntProp.
        override public int IntProp
        {
            set { m_Int = value; }
        }

        // Implement StringProp, providing a body for both the get
        // and set accessors.
        override public string StringProp
        {
            get { return (m_String); }
            set { m_String = value; }
        }
        // Declare fields to support the properties.
        private double m_Double;
        private int m_Int;
        private string m_String;
    }

    class InterfaceDemo1
    {
       static void Main(string[] args)
```

```
        {
            // Create an instance of the derived class.
            clsDerived derived = new clsDerived(3.14159);
            // Calling GetAbstract() actually calls the public method in
the
            // base class. There is no GetAbstract() in the derived
class.
            derived.GetAbstract();
            Console.ReadLine();
        }
    }
}
```

The output of the program:




2.

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

namespace Lab_guide
{
    abstract class TwoDShape
    {
        double pri_width;  // private
        double pri_height; // private
        string pri_name;   // private

        // A default constructor.
        public TwoDShape()
        {
```

```
            width = height = 0.0;
            name = "null";
        }

        // Parameterized constructor.
        public TwoDShape(double w, double h, string n)
        {
            width = w;
            height = h;
            name = n;
        }

        // Construct object with equal width and height.
        public TwoDShape(double x, string n)
        {
            width = height = x;
            name = n;
        }

        // Construct an object from an object.
        public TwoDShape(TwoDShape ob)
        {
            width = ob.width;
            height = ob.height;
            name = ob.name;
        }

        // Properties for width, height, and name
        public double width
        {
            get { return pri_width; }
            set { pri_width = value; }
        }

        public double height
        {
            get { return pri_height; }
```

```
            set { pri_height = value; }
        }

        public string name
        {
            get { return pri_name; }
            set { pri_name = value; }
        }

        public void showDim()
        {
            Console.WriteLine("Width and height are " +
                        width + " and " + height);
        }

        // Now, area() is abstract.
        public abstract double area();
    }

    // A derived class of TwoDShape for triangles.
    class Triangle : TwoDShape
    {
        string style; // private

        // A default constructor.
        public Triangle()
        {
            style = "null";
        }

        // Constructor for Triangle.
        public Triangle(string s, double w, double h) :
        base(w, h, "triangle")
        {
            style = s;
        }
```

```
        // Construct an isosceles triangle.
        public Triangle(double x) : base(x, "triangle")
        {
            style = "isosceles";
        }

        // Construct an object from an object.
        public Triangle(Triangle ob) : base(ob)
        {
            style = ob.style;
        }

        // Override area() for Triangle.
        public override double area()
        {
             return width * height / 2;
        }

        // Display a triangle's style.
        public void showStyle()
        {
            Console.WriteLine("Triangle is " + style);
        }
    }

    // A derived class of TwoDShape for rectangles.
    class Rectangle : TwoDShape
    {
      // Constructor for Rectangle.
      public Rectangle(double w, double h) :
        base(w, h, "rectangle"){ }

      // Construct a square.
      public Rectangle(double x) :
        base(x, "rectangle") { }

      // Construct an object from an object.
```

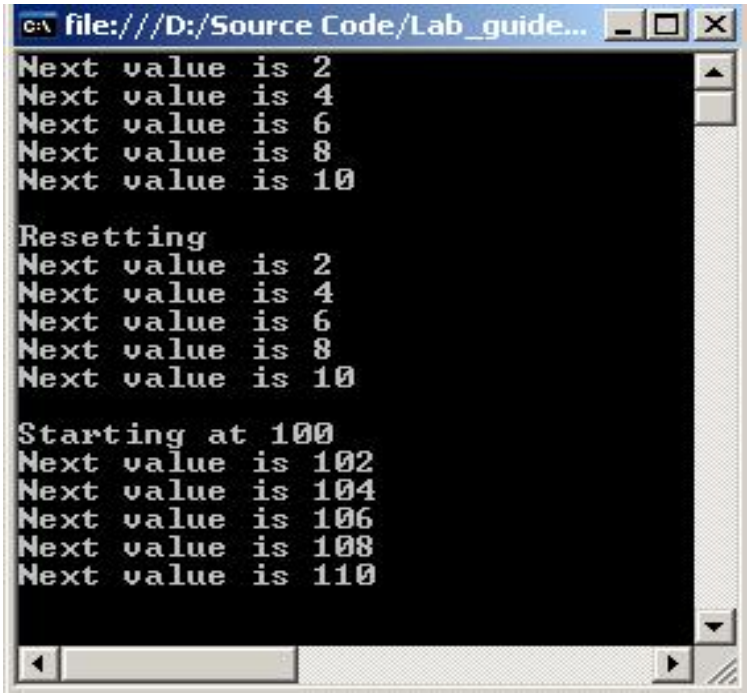
```
      public Rectangle(Rectangle ob) : base(ob) { }

      // Return true if the rectangle is square.
      public bool isSquare()
      {
        if(width == height) return true;
        return false;
      }

      // Override area() for Rectangle.
      public override double area()
      {
        return width * height;
      }
    }

    class InterfaceDemo2
    {
       static void Main(string[] args)
       {
          TwoDShape[] shapes = new TwoDShape[4];

          shapes[0] = new Triangle("right", 8.0, 12.0);
          shapes[1] = new Rectangle(10);
          shapes[2] = new Rectangle(10, 4);
          shapes[3] = new Triangle(7.0);

          for(int i=0; i < shapes.Length; i++) {
          Console.WriteLine("object is " + shapes[i].name);
          Console.WriteLine("Area is " + shapes[i].area());
          Console.WriteLine();
          Console.ReadLine();
       }
    }
}
```

**3. Creating and Using Interface.**

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

namespace Lab_guide
{
    public interface ISeries
    {
        int getNext(); // return next number in series
        void reset(); // restart
        void setStart(int x); // set starting value
    }

    // Implement ISeries.
    class ByTwos : ISeries
    {
        int start;
        int val;

        public ByTwos()
        {
            start = 0;
            val = 0;
        }

        public int getNext()
        {
            val += 2;
            return val;
        }

        public void reset()
        {
            val = start;
        }

        public void setStart(int x)
```

```
        {
            start = x;
            val = start;
        }
    }

    class InterfaceDemo3
    {
       static void Main(string[] args)
       {
           ByTwos ob = new ByTwos();

           for (int i = 0; i < 5; i++)
               Console.WriteLine("Next value is " +
                                 ob.getNext());

           Console.WriteLine("\nResetting");
           ob.reset();
           for (int i = 0; i < 5; i++)
               Console.WriteLine("Next value is " +
                                 ob.getNext());

           Console.WriteLine("\nStarting at 100");
           ob.setStart(100);
           for (int i = 0; i < 5; i++)
               Console.WriteLine("Next value is " +
                                 ob.getNext());

          Console.ReadLine();
       }
    }
}
```

The output of the program:




**4.**

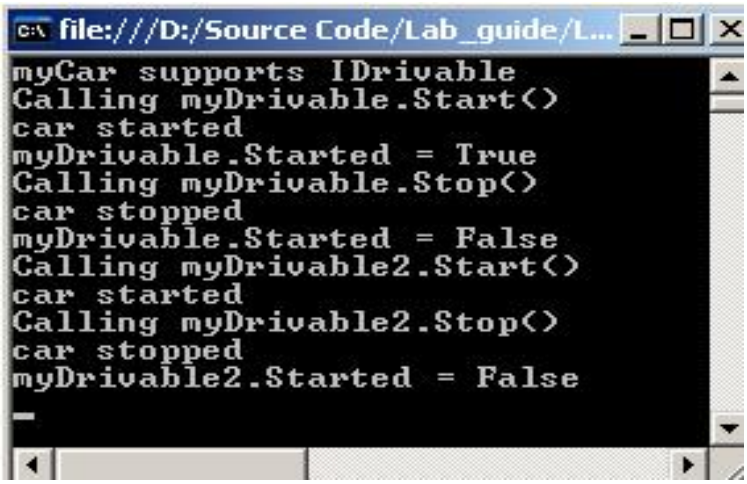
```
using System;
using System.Collections.Generic;
using System.Text;

namespace Lab_guide
{
    // define the IDrivable interface
    interface IDrivable
    {

        // method declarations
        void Start();
        void Stop();

        // property declaration
        bool Started
        {
            get;
        }
```

```
}


// Car class implements the IDrivable interface
class Car : IDrivable
{

    // declare the underlying field used by the Started property
    private bool started = false;

    // implement the Start() method
    public void Start()
    {
        Console.WriteLine("car started");
        started = true;
    }

    // implement the Stop() method
    public void Stop()
    {
        Console.WriteLine("car stopped");
        started = false;
    }

    // implement the Started property
    public bool Started
    {
        get
        {
            return started;
        }
    }
}


class InterfaceDemo4
```

```
    {
        static void Main(string[] args)
        {
            // create a Car object
            Car myCar = new Car();

            // use the is operator to check that myCar supports the
            // IDrivable interface
            if (myCar is IDrivable)
            {
                Console.WriteLine("myCar supports IDrivable");
            }

            // cast the Car object to IDrivable
            IDrivable myDrivable = (IDrivable)myCar;

            // call myDrivable.Start()
            Console.WriteLine("Calling myDrivable.Start()");
            myDrivable.Start();
            Console.WriteLine("myDrivable.Started = " +
              myDrivable.Started);

            // call myDrivable.Stop()
            Console.WriteLine("Calling myDrivable.Stop()");
            myDrivable.Stop();
            Console.WriteLine("myDrivable.Started = " +
              myDrivable.Started);

            // cast the Car object to IDrivable using the as operator
            IDrivable myDrivable2 = myCar as IDrivable;
            if (myDrivable2 != null)
            {
                Console.WriteLine("Calling myDrivable2.Start()");
                myDrivable2.Start();
                Console.WriteLine("Calling myDrivable2.Stop()");
                myDrivable2.Stop();
                Console.WriteLine("myDrivable2.Started = " +
```

```
                    myDrivable2.Started);
            }



           Console.ReadLine();
        }
    }
}
```

The output of the program:




**5. Deriving an interface from multiple interfaces.**

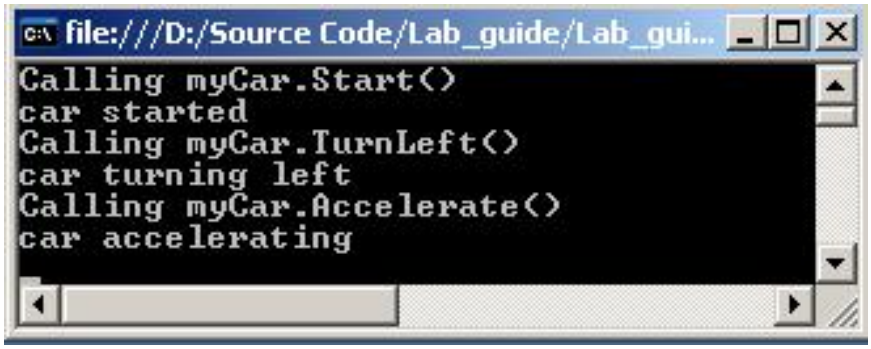
```
using System;
using System.Collections.Generic;
using System.Text;

namespace Lab_guide
{
    // define the IDrivable interface
    public interface IDrivable
    {

        // method declarations
        void Start();
        void Stop();
```

```
        // property declaration
        bool Started
        {
            get;
        }

    }

    // define the ISteerable interface
    public interface ISteerable
    {

        // method declarations
        void TurnLeft();
        void TurnRight();

    }


    // define the IMovable interface (derived from IDrivable and
ISteerable)
    public interface IMovable : IDrivable, ISteerable
    {
        // method declarations
        void Accelerate();
        void Brake();
    }

    // Car class implements the IMovable interface
    public class Car : IMovable
    {
        // declare the underlying field used by the
        // Started property of the IDrivable interface
        private bool started = false;

        // implement the Start() method of the IDrivable interface
        public void Start()
```

```
        {
            Console.WriteLine("car started");
            started = true;
        }
        // implement the Stop() methodof the IDrivable interface
        public void Stop()
        {
            Console.WriteLine("car stopped");
            started = false;
        }
        // implement the Started property of the IDrivable interface
        public bool Started
        {
            get
            {
                return started;
            }
        }
        // implement the TurnLeft() method of the ISteerable interface
        public void TurnLeft()
        {
            Console.WriteLine("car turning left");
        }

        // implement the TurnRight() method of the ISteerable interface
        public void TurnRight()
        {
            Console.WriteLine("car turning right");
        }
        // implement the Accelerate() method of the IMovable interface
        public void Accelerate()
        {
            Console.WriteLine("car accelerating");
        }

        // implement the Brake() method of the IMovable interface
        public void Brake()
```

```
        {
            Console.WriteLine("car braking");
        }

    }
    class InterfaceDemo5
    {
       static void Main(string[] args)
       {
           // create a Car object
           Car myCar = new Car();

           // call myCar.Start()
           Console.WriteLine("Calling myCar.Start()");
           myCar.Start();
           // call myCar.TurnLeft()
           Console.WriteLine("Calling myCar.TurnLeft()");
           myCar.TurnLeft();
           // call myCar.Accelerate()
           Console.WriteLine("Calling myCar.Accelerate()");
           myCar.Accelerate();
          Console.ReadLine();
       }
    }
}
```

The output of the program:

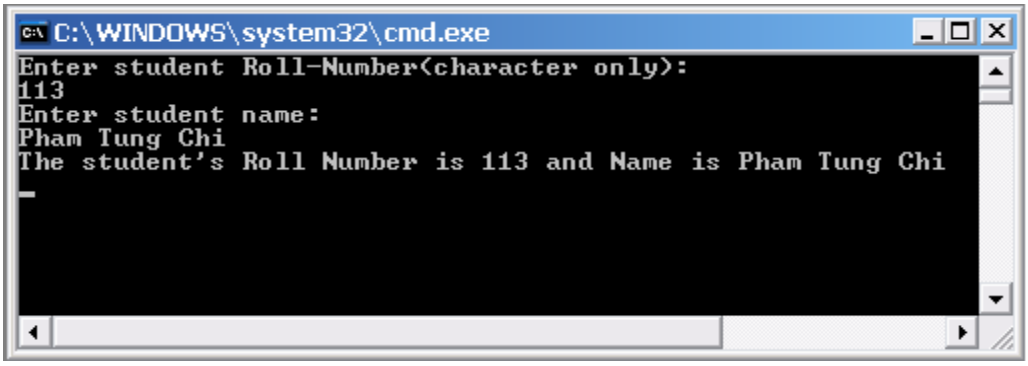



**6. Type the below code and save the file as PropertiesDemo.java**

```
using System;
```

```
public class Student
{
    public int stud_no;
    private string strName;
    public string stud_name
    {
        get
        {
            return strName;
        }
        set
        {
            strName = value;
        }
    }
}

class PropertiesDemo
{
    public static void Main()
    {
        Student objStudent = new Student();
        Console.WriteLine("Enter student Roll-Number:");
        objStudent.stud_no = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Enter student name:");
        objStudent.stud_name = Console.ReadLine();
        Console.WriteLine("The student's Roll Number is {0} and
Name is {1}", objStudent.stud_no, objStudent.stud_name);
        Console.ReadLine();
    }
}
```

The output of the program:

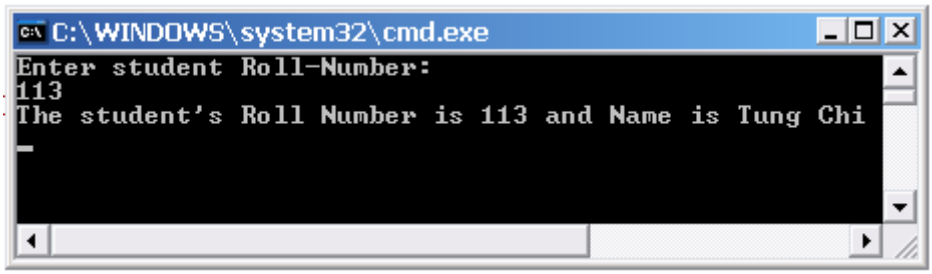



**7. Read only properties. Type below code and save to file named ReadOnlyProp.cs**

```
using System;
public class Student
{
    public int stud_no;
    public string stud_name
    {
        get
        {
            return "Ritcha";
        }
    }
}

class ReadOnlyProp
{
    public static void Main()
    {
        Student objStudent = new Student();
        Console.WriteLine("Enter student Roll-Number:");
        objStudent.stud_no = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("The student's Roll Number is {0} and
Name is {1}",
            objStudent.stud_no, objStudent.stud_name);
        System.Console.ReadLine();
```

```
   / }
}
```

The result:

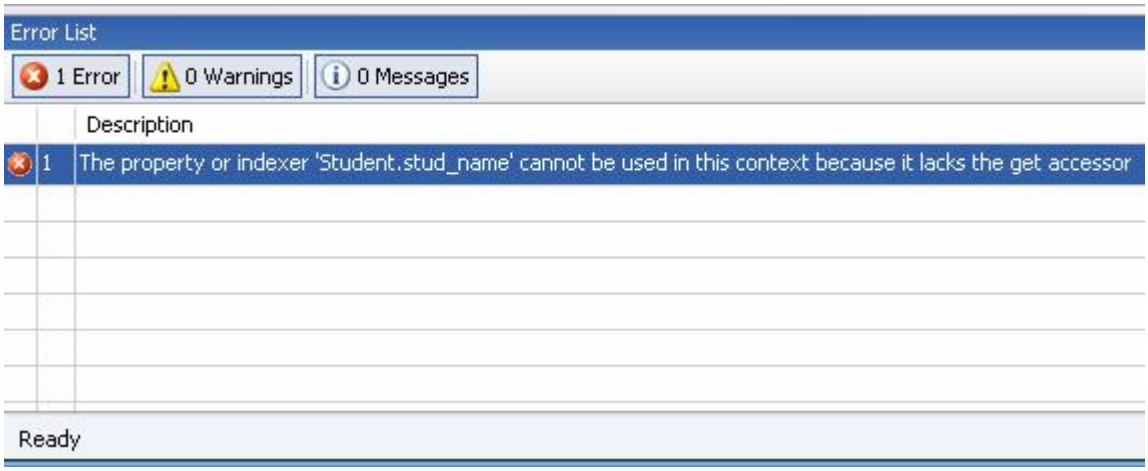



**8. Write only Properties. Type the below code and save to file named WriteOnlyProp.cs**

```
using System;
public class Student
{
    public int stud_no;
    public string strName;
    public string stud_name
    {
        set
        {
            strName = value;
        }
    }
}

class WriteOnlyProp
{
    public static void Main()
    {
        Student objStudent = new Student();
        Console.WriteLine("Enter student Roll-Number:");
```

```
        objStudent.stud_no = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Enter student Name:");
        objStudent.stud_name =
Convert.ToString(Console.ReadLine());
        Console.WriteLine("The student's Roll Number is {0} and
Name is {1}", objStudent.stud_no, objStudent.stud_name);

        System.Console.ReadLine();
    }
}
```

The result:




**9. Index Demo. Type below code and save to file named**

```
using System;

class IndexerExample
{
    public int[] intList = new int[10];

    public int this[int index]
    {
        get
```

```
        {
            return intList[index];
        }

        set
        {
            intList[index] = value;
        }
    }
}

class IndexerDemo
{
    static void Main()
    {
        int i, j = 0;
        IndexerExample indexTest = new IndexerExample();
        for (i = 1; i < 10; i += 2)
        {
            indexTest[j] = i;
            j++;
        }
        for (i = 0; i < 5; i++)
            Console.WriteLine("indexTest[{0}] is {1}", i,
indexTest[i]);
        Console.ReadLine();
    }
}
```

**10. Overloaded Index. Type below code and save to file named OvrldIndexers.cs**

```
using System;
using System.Collections;

class IndexerExample
```

```
{
    public string[] StringList = new string[10];

    public string this[int index]
    {
        get
        {
            return StringList[index];
        }
        set
        {
            StringList[index] = value;
        }
    }

    public int[,] intList = new int[10, 3];

    public int this[int index1, int index2]
    {
        get
        {
            return intList[index1, index2];
        }
        set
        {
            intList[index1, index2] = value;
        }
    }
}

class OvrldIndexers
{
    static void Main()
    {
```

```
        IndexerExample indexTest = new IndexerExample();
        indexTest[0] = "Sam";
        indexTest[0, 0] = 100;
        indexTest[0, 1] = 98;
        indexTest[0, 2] = 70;
        indexTest[1] = "Tom";
        indexTest[1, 0] = 60;
        indexTest[1, 1] = 93;
        indexTest[1, 2] = 74;
        Console.WriteLine("indexTest[1] is {0}", indexTest[0]);
        Console.WriteLine("Mark 1 of {0} is {1}", indexTest[0],
indexTest[0, 0]);
        Console.WriteLine("Mark 2 of {0} is {1}", indexTest[0],
indexTest[0, 1]);
        Console.WriteLine("Mark 3 of {0} is {1}", indexTest[0],
indexTest[0, 2]);
        Console.WriteLine("indexTest[2] is {0}", indexTest[1]);
        Console.WriteLine("Mark 1 of {0} is {1}", indexTest[1],
indexTest[1, 0]);
        Console.WriteLine("Mark 2 of {0} is {1}", indexTest[1],
indexTest[1, 1]);
        Console.WriteLine("Mark 3 of {0} is {1}", indexTest[1],
indexTest[1, 2]);
        Console.ReadLine();
    }
}
```

**Do It Yourself**

4.1. Write a program having a class **`Student`**.

a. Create a field **`stud_no`** and property **`stud_name`**. Assign and display values for the property and field.

b. Modify the class **`Student`** in exercise 1, to make **`stud_name`** as a read-only property.

c. Modify the above program to make **`stud_name`** as a write-only property.

d. Create an indexer to store odd numbers in the range of 1 to 10.

e. Create an indexer to store the names of students and overload the same to store marks of three subjects of the students.

f. Write a program to create a delegate called **`OddNumberFinder`** that displays the odd numbers in the range of 1 to 50. Create an event called **`OnOddNumber`**. This event should be fired every time an odd number is encountered and it should also display its value.

4.2. Create a namespace called **`Customer`** and add a class to it having a method that accepts customer name. Create another namespace called **`Order`** and two classes within it, one for grocery items and the other for bakery products. The `Main()` program should accept customer name and a choice indicating whether the customer has selected to order grocery items or bakery products. Accordingly, the appropriate class should be called and a message displayed informing the user about the choice.

4.3. Do the workshop 8, 9

4.3. Do ACTCSharp_Module8_Assignment.pdf in CD

4.4. Do ACTCSharp_Module9_Assignment.pdf in CD

**Reference:**


1) **MSDN Document**
2) **http://www.java2s.com/Tutorial/CSharp/CatalogCSharp.htm**
3) **CD ROM C# Programming, Aptech Computer Education**
4) **[ebook] MSDN training, Introduction to C#, Microsoft Press**