USN										

CELEBRATING 25 YEARS
CMRIT
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A+ GRADE BY NAAC

Internal Assessment Test II Scheme & Solution – OCTOBER 2018

Sub:	Dot Net Application and Framework Development				Sub Code:	15CS564	Branch:	CSE/ISE
Date:	17/10/2018	Duration:	90 min's	Max Marks: 50	Sem / Sec:	5/A,B & C		OBE

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1	Define inheritance? Explain types of inheritance with an example. Solution: Definition:-3M, Explanation:-7M	[10]	CO3	L1

Inheritance

Inheritance is the important concept of object oriented programming. It is used to avoid repetition when defining different classes that have common features and quite clearly related to one another. i.e, it provides relationship b/w the classes.

The class which inherits the members of another class is called derived class and the class whose members are inherited is called base class.

Advantages

1) Code reusability

Eg:

```
class A
{
public void Test()
 {
  Console.WriteLine ("A::Test()");
 } }
  class B : A
  {
     Public void Test1()
     {
       Console.WriteLine ("B::Test()");
     } }
     class C : B
     {
       public void Test2()
        { Console.Write ("C::Test()"); } }
```

Types of ~~Inheritance~~ program
{
Static void Main ()
{
A a = new A();
B b = new B();
~~class A~~ Base class
{
a. Test();
b. Test1();
}
class B : A
{
Test 2 ()
}
a = new B();
a. Test ()
class C : A
b = new C ();
{
b. Test();
}
Console. ReadKey().
}
}

Single — one base class & one derived class
Eg:-
class A (Base class)
{

}
class B : A (derived class)
{

}

Hierarchical — one base class is derived by many classes
Eg:-
class A
class B : A
class C : A

Multi-level — one class is derived from another derived class
Eg:-
class A
{
}
class B : A
{
}
class C : B
{
}

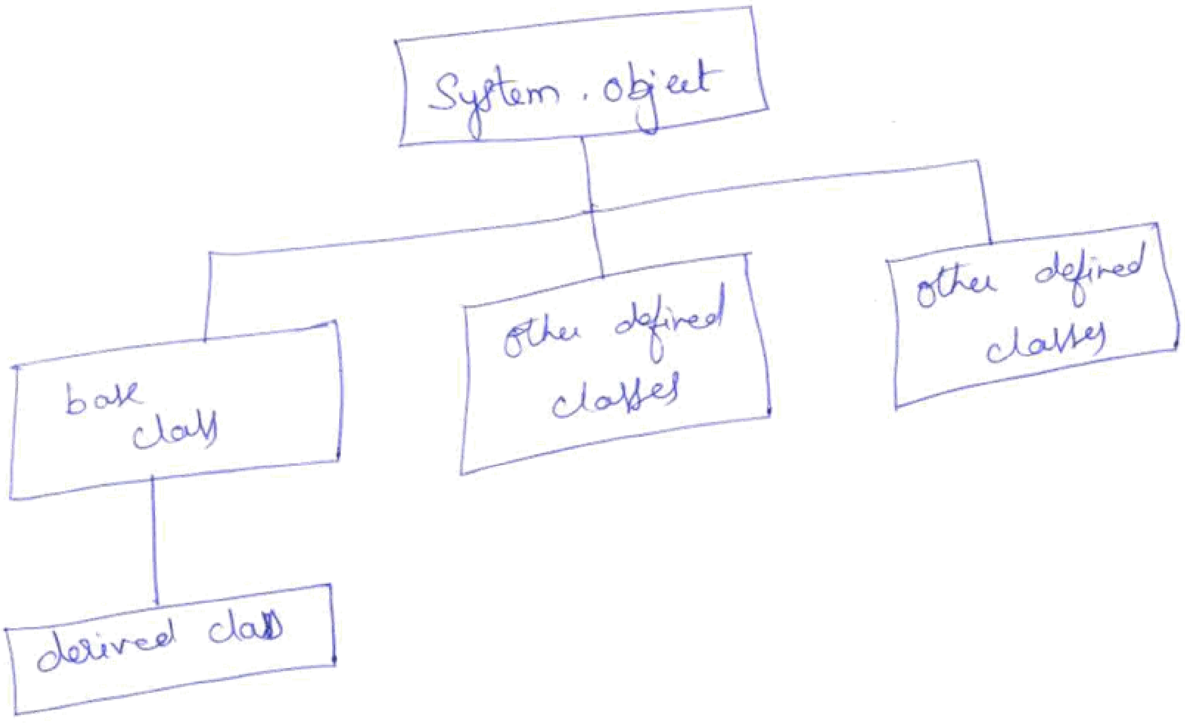
Multiple (using interface)
Eg:-
interface A
{
}
interface B
{
}
class C : A, B
{
}

Syntax for inheritance
class derived classname : Base class name

O/p :-
A : : Test ()
B : : Test ()
C : : Test ()
A : : Test ()
A : : Test ()

A class is allowed to be derived from utmost one base class that is not allowed to derive from two or more base classes unless it is sealed.

System . object is the super class which contains all the defined classes. ie, It is the root class of all classes, all classes implicitly derive from System . object



Calling base class Constructor

→ In addition to the Methods that it inherits, a derived class automatically contains all the fields from the base class.

→ When object is created these fields need to be intialized, this can be perfomed by constructor.

→ Key word "base" is used to call a base class constructor by a derived class Constructor

Eg:-
```
class shape
 {
   int x,y;
   public shape (int x, int y)
    {
      this. x = x;
      this. y = y;
 }  }
```

```
class Rectangle : shape
  {
    public Rectangle (int x , int y) : base(x,y)
    {
      this . x = x;
      this . y = y;
    }

  }
```

→ If we don't explicitly call a base class constructor in a derived class constructor, the compiler attempts to silently insert a ~~class~~ call to the base class default constructor before executing the code in the derived class constructor.

→ It works only when the base class has public default constructor

→ In case, if we don't call the correct base class constructor, the compiler will throw an error.

Eg :-

```
class shape
  {
    int x, y;
    public shape ()
    { this.x = 10;
      this.y = 100;

    }
  }
class Rectangle : shape
  { public Rectangle (int x, int y)   // No need to call explicitly
                                       the base class constructor
    {
      this . x = x;
      this.y = y; }
  }
```

2 (a)	How abstract classes are created? Explain the importance of abstract classes.	[4]	CO3	L2

Solution: Definition:-2M, Explanation about importance of abstract classes:-2M

Abstract classes

The abstract class is to provide a Common definition of a base class that multiple derived classes can share. These are similar to the interfaces except that abstract classes can contain code and data. we can specify certain methods of an abstract class as virtual.

Syntax :-

```
public abstract class <classname>
{
      class members;
}
```

Eg :-

```
abstract class Example
{
      int x, y ;
abstract void Read ();
{
   - - -
}
void display ()
{
   Console. WriteLine (x);
   Console. WriteLine (y);
 }
}
```

Advantages of Abstract Classes:

The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share. For example, a class library may define an abstract class that is used as a parameter to many of its functions, and require programmers using that library to provide their own implementation of the class by creating a derived class.

(b) Explain about parameter array.	[6]	CO3	L2

Solution: Explanation :-4M , Examples:-2M

Params

- By using params keyword we can pass any number of arguments.

Params array

- Using a params array, we can pass a variable number of arguments to a method.

Syntax: returntype methodname (params int [ ] arrayname)

Example:

```
class Example
{
        int min (params int [ ] paramList)
        {
                if (paramList == null || paramList.length == 0) throw
                        ArgumentException ("Invalid");
              minval   =   paramList   [0];
                foreach   (x   in   paramList)
                if(x<minval)
                  minval = x;
        }
        static void Main (string [ ] args)
        {
                int res1 = min (10,3);
                int res2 = min (10,2,4,6,32,80);
        }
}
```

Rules on params array

- We cannot use params keyword for multidimensional array.
  - Ex: int min (params int [,] array) //not possible
- We cannot overload a method based solely on the params keyword because params keyword is not a part of the method signature.
  - Ex: int min(params int [ ] array)
    int min(int [ ] array) //not possible
- We are not allowed to specify ref or out modifiers with params array.
  - Ex: int min(ref/out params int [ ] array) //not possible
- A params array must be the last parameter, we can have only one params array per method.
  - Ex: int min (params int [ ] array, int i) //invalid
    Int min (int I, params int [ ] array) //valid

- A non params method always takes priority over params method.
    - Ex: int min (int x, int y)
      int min (params int [ ] array)
    - If we call the above method passing only 2 parameters, the first method will be called as it gets priority.

Using params object:

- It allows us to solve the problem to pass the different number of arguments also with different types of arguments.
- If we pass a value type as a parameter, the compiler generates the code to convert the value type to reference type by boxing.

Example:

```
class Example
{
        static void AnyMethod (params object [ ] paramList)
        {
                if (paramList == null || paramList.length == 0) throw
                        new ArgumentException (“Invalid”);
                foreach (object x in paramList)
                        Console.WriteLine(x);
        }
        static void Main(String [ ] args)
        {
                AnyMethod (null);                          // null value(exception)
                AnyMethod ( );                             // length = 0(exception)
                int [ ] a = {10, 5, 7, 20};
                AnyMethod (a);                             // boxing
                AnyMethod (“Appu”, 24, “Idly”, 80.56);     // boxing for int values
        }
}
```

Difference between optional parameters and params array

- Optional parameters
    - A method that takes optional parameters, still it has a fixed parameter list and we cannot pass an arbitrary list of arguments.
    - The compiler generates code that inserts the default values onto the stack for any missing arguments before it runs.
    - Its not aware of which of the arguments are provided by caller and which are compiler generated defaults.
- Params Array
    - A method that uses the params array efficiently as a complete arbitrary values and none of them are default.
    - The method will determine exactly how many arguments the caller provided.

3	Define Interface? Explain how interfaces are created with an example.	[10]	CO3	L1

<u>Solution</u>: Explanation :-8M , Examples:-2M

Interface

- It does not contain any code or data. It just specifies the methods and the properties that a class that inherits from an interface must provide.
- By using an interface, we can completely separate the names and the signatures of the method of a class from methods implementation.
- Syntax is similar to defining a class except that we have to use interface keyword.
- Never specify an access modifier like public, private or protected because interface by default is public.
- All the methods in the interface should be declared but not to be implemented inside the interface.
- The classes which implements the interface will define all the methods declared in the interface.

Example:

```
interface ILandBound
{
        int NoOfLegs ();
}
class Horse : ILandBound
{
        int NoOfLegs ()
        {
                return 4;
        }
}
```

Note:

- If we want to inherit base class with interface, we have to specify the base class name followed by any number of interfaces.

Example:

```
interface ILandBound
{
    int NoOfLegs ();
}
class Mammal
{
    ….
    ….
}
class Horse : Mammal, ILandBound
{
    int NoOfLegs ()
    {
        return 4;
    }
}
```

- A class can inherit only one base class but n number of interfaces.
- Always the first parameter after the ":" should be the base class and then followed by n number of interfaces.

Example:

```
class Horse : Mammal, ILandBound
{
……
……
}
```

- Rules:
    - The method name and the return type should match exactly.
    - In parameter including ref and out keyword modifier must match exactly.
    - All methods implementing an interface must be publicly accessible. However if we are using an explicit interface implementation, methods should not have access specifier.
    - If there is any difference between interface definition and declaration, compiler shows an error.

- Referencing a class through its interface:
    - We can assign the reference of a class to the reference of an interface if and only if the class is implementing that interface. Else the runtime will throw an InvalidTypeCastException.
    - Converse is not possible i.e., we cannot assign interface reference to the class reference.
    - To typecast safely, we need to use is and as keywords.

```
interface ILandBound
{
        int NoOfLegs();
}
class Horse : ILandBound
{
        int ILandBound.NoOfLegs()
        {
                return 4;
        }

        static void Main(string [ ] args)
        {
                Horse h=new Horse(); ILandBound
                i=h;                            //legal

                IJourney j=h;                   //invalid cast exception
                //So for this
                if(h is ILandBound)
                {
                        ILandBound i=h;
                }
                 //or
                        ILandBound i=h as ILandBound;
                }
        }
}
```

- WORKING WITH MULTIPLE INTERFACES
  - A class can have at most one base class but it can implement any number of interfaces.but it should implement all the methods declared by there interface.
  - Many interface will declare aa method with the same name,if a class implementing this interface the method definitions belongs to both the interface.

Example:

```
interface ILandBound
{
        int NoOfLegs();
}
interface IJourney
{
        int NoOfLegs();
}
class Horse : ILandBound, IJourney
{
        int ILandBound.NoOfLegs()          //this implementation belongs
                                           to both the interface
        {
                return 4;
        }

}
```

  - To avoid this will go for EXPLICITLY IMPLEMENTING AN INTERFACE: specify which interface a method belongs to when we implement it.

```
Syntax: return type interface name.Methodname()
{
…………
………….
}
```

Example:

```
class Horse : ILandBound, IJourney
{
        int ILandBound.NoOfLegs()
        {
                return 4;
        }
        int IJourney.NoOfLegs()
        {
                return 3;
        }
}
```

Restriction of Interface:

- In an interface we are not allowed to define any fields even if it is static field.
- Interface is not allowed to declare any constructor or destructor because it requires implementation. For this purpose interface instance cannot be reated.
- Cannot specify any access modifier for the interface as well as method declaration inside the interface because implicitly it is public.
- We cannot declare or define any enumeration.

- Note:
  - Interface is not allowed to inherit from structures or a class but interface can inherit from another interface.
  - For example if interface A inherits(interface extension) from interface B any class which implements interface A must and should provide implementation for interface B too.

Example:

```
interface ILandBound
{
        int NoOflegs ();
}
interface IJourney : ILandBound
{
        int NoOfJourney ();
}
class Horse : IJourney
{
        int NoOfLegs()
        {
                return 4;
        }
        int NoOfJourney()
        {
                return 3;
        }
}
```

4	Write a c# program for create a windows application to build the Phone Book with following options using Indexers i) Add information ii) Search by Phone number iii) Search by Name .	[10]	CO4	L3

<u>Solution</u>: Program:-10M

1. Create a form ,displaying two empty text boxes labeled **Name** and **Phone Number.** Two buttons one to find a phone number when given name and one to find a name when given a phone number. And **Add button** that will add a name/phone number pair to a list of names and phone numbers held by the application.
2. Write the indexers to perform search operations

```
Sealed class Phonebook
{
   Public Name this [Phonebook number]
{
```

```
Get
{
       int i=Array.Indexof(this.phonenumbers,number);
if(i!=-1)
{
     return this.names[i];;
}
else
{
      return new Name();
}}}
   Public PhoneNumber this [Name name]
{
Get
{
       int i=Array.Indexof(this.names,name);
if(i!=-1)
{
     return this.phoneNumbers[i];;
}
else
{
      return new PhoneNumber();
}}}
Private void findbynameclick(object sender,RoutedEventArgs e)
{
   String text = name.Text;
If(!string.IsNullorEmpty(text))
{
  Name personName = new Name(text);
  PhoneNumber personPhoneNumber=this.phonebook[personName];
  PhoneNumber.Text = String.IsNullorEmpty(personsPhoneNumber.Text)?”not
found”:personphonenumber.Text;
}
}
Private void findbyPhoneNumberclick(object sender,RoutedEventArgs e)
{
   String text = phonenumber.Text;
If(!string.IsNullorEmpty(text))
{
  PhoneNumber personPhoneNumber = new PhoneNumber(text);
  Name personName=this.phonebook[personPhoneNumber];
 Name.Text=String.IsNullorEmpty(personName.Text) ? “not found”: personName.Text;
}
}

}
```

5	Why use the garbage collector? Explain with example and Explain How does the garbage collector work with recommendations?	[10]	CO3	L2

Solution: Use of garbage collector:-2M ,Explanation:-8M

The Common Language Runtime (CLR) will call the destructor.

- The lifetime of an object

We can create the object by using new operator. Ex:
Square s = new Square ();
We think that new operator is a single step process, but it has

- The new operator allocates a chunk of raw memory from the heap. The user doesn't have any control over this phase.
- The new operation converts the chunk of raw memory into an object, it has to initialize the object. The user can control this phase by using constructor.

After an object is created we can access its members using the dot operator. Ex: s.Area ();

- Destroying the object

When the object variable goes out of scope, the object is no longer being actively referenced.
Then the object can be destroyed and the memory that it is using can be reclaimed.
The object destruction is a two phase process:

The Common Language Runtime (CLR) must perform some tidying up. User can control this by destructor.

Once the destructor is called by CLR, it must return the memory previously belonging to the object back to the heap. The process of destroying an object and returning the memory back to the heap is known as garbage collector.

- Destructor

Destructor is a special method quite same like constructor except that CLR calls it after the reference to the object is disappeared.

Syntax:
```
~NameOfClass ()
{
    ……
    ……
}
```

- Destructor is used to perform any tidying up that requires when object is garbage collected.
- The CLR automatically clears up and manages the resources, so defining destructor is not mandatory.
- If managed resources is large or the object is referring to null reference, then defining the destructor is useful.

- Rules for defining the destructor
  - Destructor applies to reference type. We cannot define destructor for value type like struct.
  - We cannot specify an access modifier such as public for a destructor because CLR calls it automatically.

Example:
```
public ~Square ()
{
    ……
    ……
}
```

- Destructor doesn’t take any parameters because we cannot call the destructor.

Example:
```
public ~Square ()
{
        ……
        ……
}
```

File Processing

```
class FileProcessor
{
        FileStream file = null;

        public FileProcessor (string name

        {
                this.file = OpenRead (name);

        }
        ~FileProcessor ()

        {
                this.file.Close();
        }
}
```

Internally after C# compiles this code, automatically it translated destructor into an override of Object.Finalize() method as shown below.

Compiler Code:

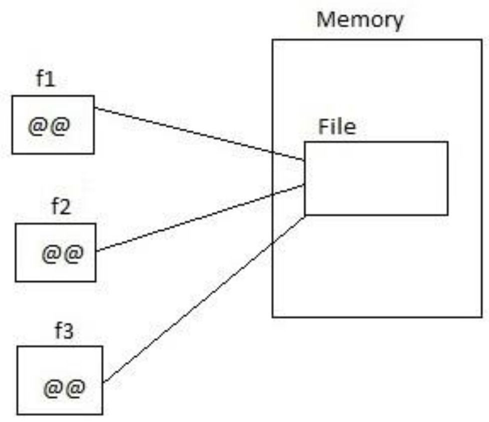
```
class FileProcessor
{
        protected override void Finalize ()
        {
                try
                {
                        ……
                        ……
                }
finally
{


base.Finalize ();
}
}
}
```

To ensure that destruction always calls its base class destructor and exception can occur in destructor also, so whatever code we have put as destructor, the compiler puts it in the try block.

- Why to use the garbage collector?

We cannot destroy an object by ourselves by using C# code.
CLR does it for us at the time of its own choosing.
In C# more than one reference variable refers to same object and any number if reference variable can refer the same object. So CLR keeps track of all these references.
Ex:



If f1 disappears, the object still exists in the heap. We cannot reclaim the space.
So lifetime of an object cannot be tied to a particular reference variable.
An object can be destroyed and its memory made available only when all the reference variables disappear.

- Why is the responsibility not given to the user?

We might forget to destroy the object. Destructor will not run. So tidying up would not occur and memory is not returned back to heap. The any user will not be able to use that memory. It will be a waste.
We try to destroy an active object and the risk is that one/more reference variable hold a reference to a destroyed object. It refers to either unused memory or possibly to a completely different object that happens to occupy same block of memory.
We may try to destroy same object more than once.

- Goals of garbage collector

Every object will be destroyed and its destructor will be run when the program end and all the outstanding objets will be destroyed.
Every object will be destroyed exactly once.
Every object will be destroyed only when it becomes unreachable, ie, no reference variable is pointing to that object.

- How the garbage collector works

The garbage collector runs in its own thread and can execute only at certain time typically when the application reaches the end of the method.
When garbage collector is running, other thread will be halted because garbage collector might need to move the object around and update object reference.
It has the following stpes:

➢ It builds a map of all reachable objects. It does this by reeatedly following references fields inside objects.

- It maps very carefully and ensures that circular deferences will not point to infinite recurssion.
- It checks for unreachable objects that has a destructor that need to be run before finalization and places all these objects in the FReachable queue.
- It deallocated the unreachable objects that doesn't require finalizaation and reclaims the memory back to heap. At this point it allows other threads to resume.
- It finalizes the objects that requires Finalization by running

- Recommendation

Writing destructor adds complexity to our code and the garbage collector process and makes programs runs slowly. So only if we have unmanaged resources to define the resources.

6	Briefly explain the properties in C#.	[10]	CO4	L1

Solution: Explanation:-10M

Properties

Syntax:

```
accessModifier type PropertyName
{
          set
        {
                  …..
        }
          get
        {
                  …..
        }
}
```

Example:

```
struct Screenposition
{
        private int x, y;
        public int X
        {
                  set
                {
                        this.x = checkRange (value);
                }
                  get
                {
                        return this.x;
                }
        }
        public int Y
        {
                  set
                {
                }       this.y = checkRange (value);
        }
```

```
                    private static int checkRange (int value)
                    {
                              if (value < 0 || value > 1024)
                              {
                                        throw new ArgumentOutOfRangeException;
                              }
                              return value;
                    }
                    static void Main (String [ ] args)
                    {
                              int value;
                              ScreenPosition st = new ScreenPosition ();
                              s.X = 100;
                              value = s.X;
                              s.X+= 10;
                    }
          }
```

Property:

- It is a cross between a field and a method.
- It looks like a field but acts like a method.
- User will access the same like a field but the compiler will take as a method.
- A property can contain two blocks of code which starts with set and get keywords.

Using Property

```
s.x = 100;                                   //set
value = s.x;                                 //get
s.x += 10;                                   //both set and get
```

Read only property

```
accessModifier type PropertyName
{
           get
          {
                    …..
                    …..
          }
           }
```

Write only property

```
accessModifier type propertyName
{
 set
{
          …..
          …..
}
 }
```

Property Accessibility

- We can change the accessibility of only one of the accessor when we define it (get or set).
- It would not make much sense to define a property as public only to change the accessibility of both accessor to private anyways.
- The modifier must not specify an accessibility that is less respective than that of the property. (Don't make the property as private. Instead make either get or set as private.)

```
struct ScreenPosition

{

          private int x;

          public int X
          {
                    private set
                    {
                              this.x = checkRangex (value);
                    }
                    public get
                    {
                              ……
                              ……
                    }
          }

          void change ()

          {
                    x = 100;

          }
}
```

Using property appropriately:

- Properties are powerful features but we have to use in the correct manner. Ex:

```
class Account
{
```

```
private int bal;
public int Balance
{
     set
    {
          this.bal = value;
    }
     get
    {
          return this.bal;
    }
}
 }
```

- In the above example, the balance of the user is not updated correctly. Suppose a person has Rs. 1000 and he uses get property to withdraw Rs. 500, the Rs. 1000 balance is not changed to Rs. 500.
- So this is a poor programming method.
- To overcome this we use separate withdraw and deposit methods.and give only read permission for balance property.

```
public int Balance { get{return this.bal; } }
```

Property restrictions:

- We can assign a value through a property of a structure or a class only after structure of class variable is been initialized (object is created).
- Without object, we get error. Ex:

```
ScreenPosition s;
s.x = 100;                                  //Error

ScreenPosition s = new ScreenPosition ();

s.x = 100;                                  //Valid
```

- We cannot use a property with ref or out arguments to a method. Ex: void method (ref s.x)   //Error

- Property can contain atmost one get accessor and one set accessor. It cannot contain other methods or fields or properties.

- The get and set cannot take any parameters. The parameters being assigned is passed to the set accessor automatically by using value variable.

- We can't declare property using const and static, but we can have if condition in get and set accessor.

Declaring interfaces properties

- Interface can declare properties as well as methods.
- A class or a structure that implements this interface must implement the get as well as set accessors for the properties.
- If the class implements the interface, we can declare the property implementation as virtual which enables the derived classes to override the implementation.
- If we implement an interface in a class, that definition belongs to a class as well as interface. If we want to give a specific implementation for interface, we need to go for explicit interface definition.

  Ex:

```
interface IScreenPosition
{
     int x
    {
        get;
        set;
    }
     int y
    {
        get;
        set;
    }
}
class Example : IScreenPosition
{
    private int _x, _y;
    public int x
    {
         get
        {
            return this._x;
        }
         set
        {
            this._x = value;
        }
    }
    int IScreenPosition.X
    {
        get
```

```
                        {
                                return this._x;
                        }
                         set
                        {
                                if (value < 1024)
                                        this._x = value;
                        }
                }
                Main ()
                {
                        Example c = new Example ();
                        c.X= 10;
                        Console.WriteLine (c.X);
                        IScreenPosition i = e as IScreenPosition (); if (i
                        != null)
                        {
                                i.x = 100;
                                Console.WriteLine (i.X);
                        }
                }
        }
```

Generating automatic property

- We need to focus on two things with respect to properties
  - Compatibility with application: The fields and properties explore themselves by using different metadatas in assembly. If fields are public, it can be accessed by any class but properties enforces encapsulation. If we want to change a field to a property, entire application changes and we need to recompile the application.

  - Compatibility with interfaces: If we are implementing an interface, and the interface defines an item as a property, the class must and should write a property that matches the specification to the interface. We cannot implement a property for public fields.

- Because programmers are busy, the compiler automatically generates the code for property for the programmer's convenience.

- It looks like a interface but here we specify access modifiers like public, private or protected.

- We can have automatic get but not set.for example we should not change the file when it is created or owner of the file.

- For this reason, we need to initialize the value. Ex:

```
public DateTime day
{
        get;
} = DateTime.now;
class Example
{
        private DateTime day;
        public Example (DateTime y0
        {
                Day = y;
        }
}
```

Initializing Objects by using properties

- For various scenarios or combinations, we must initialize the fields. We will end up by writing multiple constructors.
- We are not able to write a unique constructor which has same time and same number of parameters.
- To overcome this drawback:
  - Use optional parameters and pass the arguments in the constructor by using named arguments.
  - Better and transparent solution is using properties. Initialize the private fields to set a default values and expose them as properties.

Ex:

```
class Triangle
{
        private int s1 = 10, s2 = 10, s3 = 10;
        public int Side1
        {
                 get
                {
                        return this.s1;
                }
                 set
                {
                        this.s1 = value;
                }
        }
        public int Side2
        {
            get
            {
                return this.s2;
            }
            set
            {
                this.s2 = value;
            }
        }
        public int Side3
        {
            get
            {
                return this.s3;
            }
            set
            {
                this.s3 = value;
            }
        }
         }
static void Main (String [ ] args)
{
        Triangle t1 = new Triangle { Side1 = 15 };          //this is called object initializer
        Triangle t2 = new Triangle { Side3 = 15, Side 2=12 };
}
```

- When we invoke object initializer, C# generates the code for calling the default constructor and then it calls the set accessor of the property.

- We can also call a parameterized constructor and then the object initializer.

- Remember that always the constructor will be called first and then the property.

  Write a C# program which defines a class Polygon which has 2 properties number of sides and side length. Using object initializer, create 3 types of polygons Square, Triangle, Pentagon and print each polygon’s sides and its values.

```
class Polygon

{

        String name;
        private int sides = 10, length = 10;

        public Polygon (String name)


        {
                this.name = name;

        }

        public int Side

        {
                 get
                {
                        return this.sides;

                }
                 set
                {

                }       this.sides = value;

        }

        public int Length
        {
                 get
                {


                }

                 Set

                {

                }
        }

         }

return this.length;



this.length = value;
```

```
public class Demo
{
        static void Main (String [ ] args)
        {
                Polygon p1 = new Polygon ("Square") { Side = 4, Length = 20 }; Polygon
                p2 = new Polygon ("Triangle") { Side = 3, Length = 10 }; Polygon p3 =
                new Polygon ("Pentagon") { Side = 5, Length = 40 }; Console.WriteLine
                (p1.name + " " + p1.Side + " " + p1.Length); Console.WriteLine (p2.name,
                p2.Side, p2.Length); Console.WriteLine (p3.name, p3.Side, p3.Length);
        }
}
```

Output:

```
Square 4 20
Triangle 3 10
Pentagon 5 40
```

7	Define indexers? Explain the use of indexers.	[10]	CO4	L1

Solution : Definition:-2M ,Explanation:-8M

C# provides a set of operators that we can use to access and manipulate the individual bits like NOT operator, Left Shift, Right Shift, OR, etc.

Suppose for a binary number, we need to check if the 6th bit from right is 0 or not. We use bits = bits & (1<<5) != 0

To set bit to 0

bits bits & ~(1<<5)

To set bit to 0

bits = bits 1 (1<<5)

Unfortunately this is not an abstract solutions because we need to repeat this logic to set or unset for specific index.

For that purpose we can't give square bracket notation for integer because it works only for arrays.
We can't give

bits [3] = true;

bits [5] = false;

We cannot use [ ] notation in int type variables.

So we need to create a new type that acts like and use like arrays of bool variables but implemented for int. That is knows as indexer.

Indexer:

- Indexer is same like property as smart field where a property encapsulates a single value in a class but an indexer encapsulates a set of values.
- An indexer is not a method. There is not parenthesized containing a parameter but there are square brackets specifying the index.
- All indexer uses this keyword.
- A class or a structure can use at most 1 indexer or it can overload an indexer but always the name of the indexer is this.
- Indexer acts like a property which has get and set accessor.
- The index specified in the indexer is populated when the indexer is called.

  Ex:

```
        struct IntBIts

        {

                private int bits; public

                intBits(int val)

                {

                        bits = val;

                }

                 public bool this[int index]
```

```
              {
              get
              {
                              return (bits &(1<<index) != 0);
              }
              set
              {
                              if(value)
                                   bits = bits|(1<<index);
              else
                                   bits = bits&(1<<index);
              }
       }
                   public void display()
                   {
                        Console.WriteLine("bits = {0}", bits);
                   }
               }
               Class program
               {
                   Static void Main(string [ ] args)
                   {
                        IntBits b = new IntBits(126);
                        bool ans = b [6]                    //retrieves bool at index 6;
                        b[0]   =   true;
                        b[3] = false;
                        b.display ();                       //ans will be 119;
                   }
               }
```

UNDERSTANDING INDEXER ACCESSOR

- When we read an indexer the compiler automatically translates the array like code into call to get accessor of that indexer.

```
               bool p = b [5];                    //get
```

- When we write to an indexer the compiler automatically translates the array like code into call to set accessor of that indexer.

```
                b [1] = true;                     //set
```

- It is also possible to use an indexer in a combined read/write context b [5] =
               b[5]^true;   //set and get

program to show how array is implemented using indexer.

```
 class IndexerDemo
      {
           private String [ ] nameList = new String [10];
           public String this [this Index]
           {
          get
          {
                         if (Index >= 0 && Index < 10)
                              return nameList [Index];
          }
          set
          {
                         if (Index >= 0 && Index < 10)
                              nameList [Index] = value;
          }
    }
           static void Main (String [ ] args)
           {
                IndexerDemo ID = new IndexerDemo ();
                ID[0] = "Piddy";

                ID[5] = "Pradeep";

                for (int i = 0; i < ID.length; i++)
                     Console.WriteLine (ID [i]);
           }
      }
```

Comparing indexers and arrays:

- Indexers can use non numerical subscript like String, char, float, double.But arrays can use only integers as subscript.

- Indexers can be overloaded just like methods public

```
int this [String Index]
{
     ….
     ….
}
public int this [String Index]
{
     ….
     ….

}
```

- Indexers cannot be used as ref and out as parameters because it already is a reference type.
  Method(ref bits[5]);

Comparison between property array and indexer array Property array

- It is possible that the property can return an array but since the arrays are reference types so exposing an array as a property creates the possibility of accidently overwriting the data.

```
class PropertyDemo
{
        private  int  [  ]  data  =  new  int  [10];
        public int [ ] Data
        {
       set
       {
                        this.data [i] = value;
       }
       get
       {
                        return this.data [i];
       }
 }
        static void Main (String [ ] args)
        {
                PropertyDemo PD = new PropertyDemo ();
                int [ ] value = PD.Data;//reference to reference copy value [0] ++;//
                its also updates in private field data[0] too.
        }
}
```

Indexer Array

```
class IndexerDemo
{
        private  int  [  ]  data  =  new  int  [10];
        public int this[int i]
        {
                set
                {
                        this.data [i] = value;
            }
            get
            {
                        return this.data [i];
            }
    }
             static void Main (String [ ] args)

             {

                    IndxerDemo ID = new IndexerDemo ();
                     int [ ] value =new int[5];
                    value [0] =ID[0];it just assignes the value not the reference
                    value[0]++//upadtes only in value array not in private array data[0]
             }
      }
```

Indexers in Interface

- We can declare indexers in an interface specifying the get keyword and set keyword or but replace the body of get and set accessor with a semicolon.

  Any class or structure implements the interface must implement the indexer accessors declared in the interface.

```
interface IData
{
        int this [int Index]

        {

                get;
                set;
        }
}

class Example : IData
{

        private int [ ] data = new int [10]; int

        IData. this [int index]
        {
```

```
                        set
                        {

                        }
                        get
                        {


  this.data [i] = value;
return this.data [i];
}
}
}
```