Object Oriented Programming(OOP)

The programming in which data is logically represented in the form of a class and physically represented in the form an object is called as object oriented programming (OOP). OOP has the following important features.

Class : In OOP languages it is must to create a class for representing data. Class contains variables for storing data and functions to specify various operations that can be performed on data. Class will not occupy any memory space and hence it is only logical representation of data.

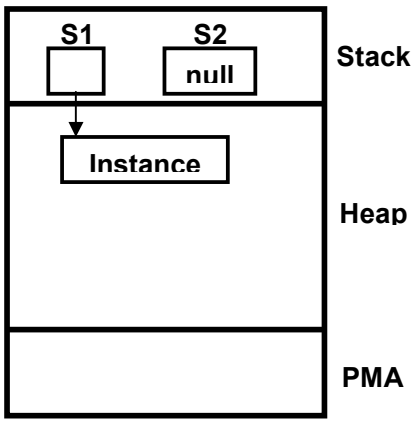
Data Encapsulation : Within a class variables are used for storing data and functions to specify various operations that can be performed on data. This process of wrapping up of data and functions that operate on data as a single unit is called as data encapsulation.

Data Abstraction : Within a class if a member is declared as **private,** then that member can not be accessed from out side the class. I.e. that member is hidden from rest of the program. This process of hiding the details of a class from rest of the program is called as data abstraction. Advantage of data abstraction is security.

Object And Instance : Class will not occupy any memory space. Hence to work with the data represented by the class you must create a variable for the class, which is called as an object. When an object is created by using the keyword **new**, then memory will be allocated for the class in heap memory area, which is called as an instance and its starting address will be stored in the object in stack memory area.

When an object is created without the keyword new, then memory will not be allocated in heap I.e. instance will not be created and object in the stack contains the value **null.** When an object contains null, then it is not possible to access the members of the class using that object.

Student S1=new Student();
Student S2;



Inheritance : Creating a new class from an existing class is called as inheritance. When a new class requires same members as an existing class, then instead of recreating those members the new class can be created from existing class, which is called as inheritance. Advantage of inheritance is reusability of the code. During inheritance, the class that is inherited is called as base class and the class that does the inheritance is called as derived class.

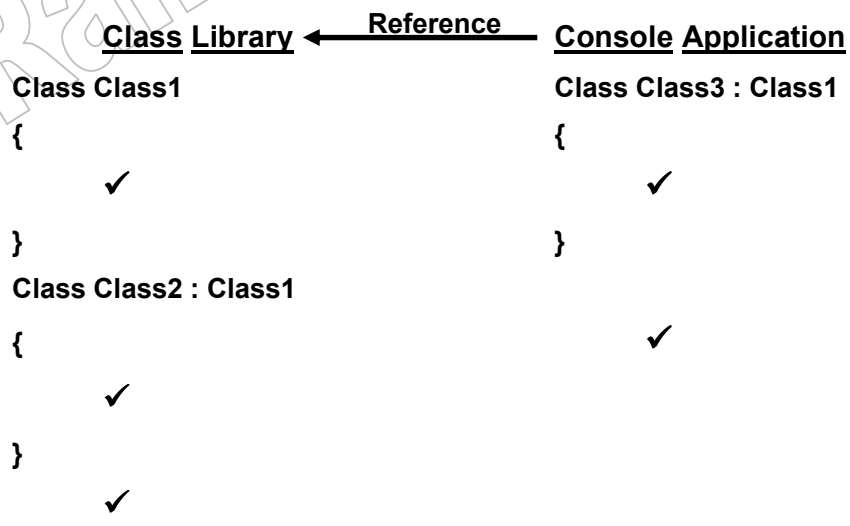
Polymorphism : Polymorphism means having more than one form. Polymorphism can be achieved with the help of overloading and overriding concepts. Polymorphism is classified into compile time polymorphism and runtime polymorphism.

Access Modifiers

An access modifier is a keyword of the language that is used to specify the access level of members of a class. C#.net supports the following access modifiers.

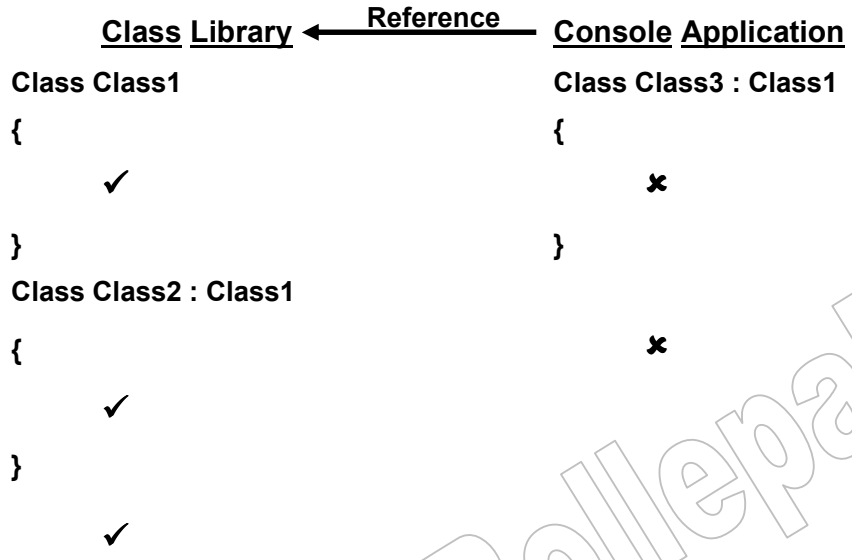
Public : When Members of a class are declared as public, then they can be accessed

1. Within the class in which they are declared.
2. Within the derived classes of that class available within the same assembly.
3. Outside the class within the same assembly.
4. Within the derived classes of that class available outside the assembly.
5. Outside the class outside the assembly.



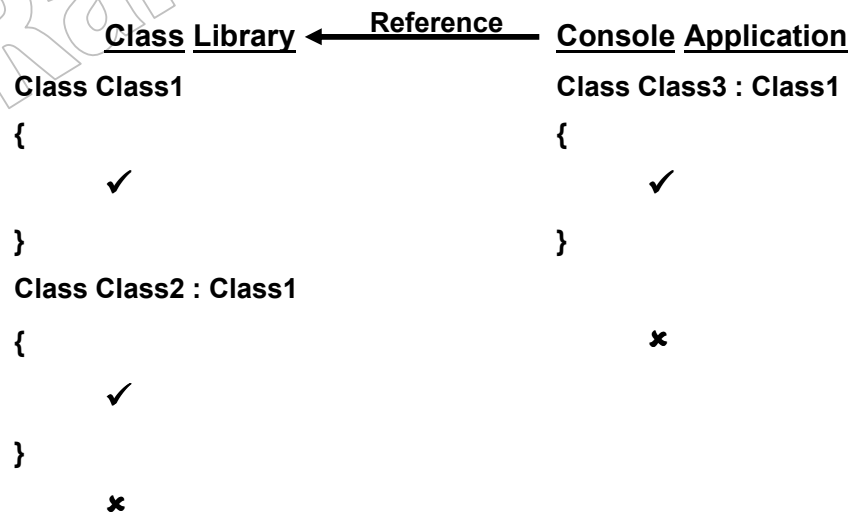
Internal : When Members of a class are declared as internal, then they can be accessed

1. Within the class in which they are declared.
2. Within the derived classes of that class available within the same assembly.
3. Outside the class within the same assembly.



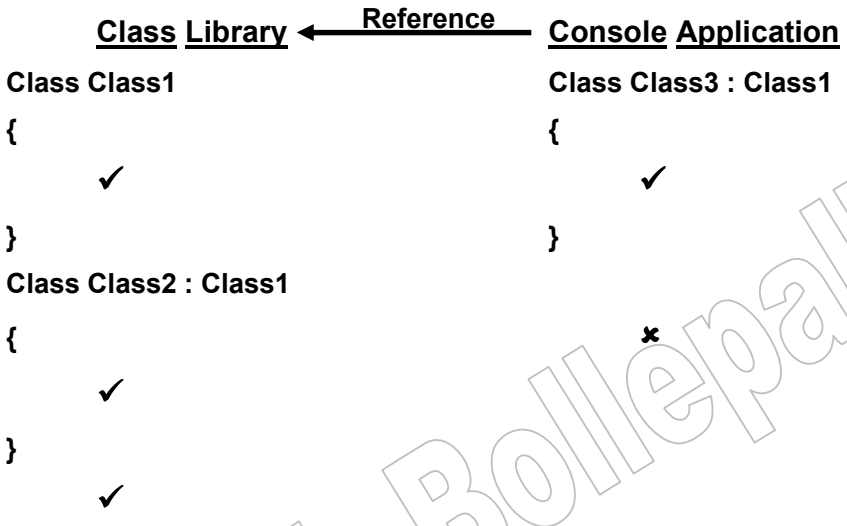
Protected : When Members of a class are declared as protected, then they can be accessed

1. Within the class in which they are declared.
2. Within the derived classes of that class available within the same assembly.
3. Within the derived classes of that class available outside the assembly.



Protected internal : When Members of a class are declared as protected internal, then they can be accessed

1. Within the class in which they are declared.
2. Within the derived classes of that class available within the same assembly.
3. Outside the class within the same assembly.
4. Within the derived classes of that class available outside the assembly.



Private : Private members of a class are completely restricted and are accessible only within the class in which they are declared.

Creating Classes

To create a class, use the keyword class and has the following syntax.

```
[Access Modifier] class ClassName
{
    -
    -
    -
}
```

A class can be created with only two access modifiers, public and internal. Default is public. When a class is declared as public, it can be accessed within the same assembly in which it was declared as well as from out side the assembly. But when the class is created as internal then it can be accessed only within the same assembly in which it was declared.

Members of a Class

A class can have any of the following members.

Fields

Properties

Methods

Events

Constructors

Destructor

Operators

Indexers

Delegates

Fields : A field is the variable created within the class and it is used to store data of the class. In general fields will be private to the class for providing security for the data

Syntax : [Access Modifier] DataType Fieldname;

Properties : A property is a method of the class that appears to the user as a field of the class. Properties are used to provide access to the private fields of the class. In general properties will be public.

```
Syntax :    [Access Modifier] DataType PropName
            {
                Get
                {
                }
                Set
                {
                }
            }
```

A property contains two accessors, get and set. When user assigns a value to the property, the set accessor of the property will be automatically invoked and value assigned to the property will be passed to set accessor with the help of an implicit object called **value**. Hence set accessor is used to set the value to private field. Within the set accessor you can perform validation on value before assigning it to the private field.

When user reads a property, then the get accessor of the property is automatically invoked. Hence get accessor is used to write the code to return the value stored in private field.

Readonly Property : There may be a situation where you want to allow the user to read the property and not to assign a value to the property. In this case property has to be created as readonly property and for this create the property only with get accessor without set accessor.

Syntax : **[Access Modifier] DataType PropName**

{

Get

{

}

}

Writeonly Property : There may be a situation where you want to allow the user to assign a value to the property and not to read the property. In this case property has to be created as writeonly property and for this create the property only with set accessor without get accessor.

Syntax : **[Access Modifier] DataType PropName**

{

Set

{

}

}

Methods

Methods are nothing but functions created within the class. Functions are used to specify various operations that can be performed on data represented by the class.

Example : The following example creates a class with the name Marks for accepting marks in three subjects and calculate total, average and grade for the marks.

```
namespace OOPS
{
    class Marks
    {
        int _C, _Cpp, _DotNet, _Total;
        float _Avg;
        string _Grade;
```

```
    public int C
    {
        get { return _C; }
        set { _C = value; }
    }
    public int Cpp
    {
        get { return _Cpp; }
        set { _Cpp = value; }
    }
    public int DotNet
    {
        get { return _DotNet; }
        set { _DotNet = value; }
    }
    public int Total
    {
        get { return _Total; }
    }
    public float Avg
    {
        get { return _Avg; }
    }

    public string Grade
    {
        get { return _Grade; }
    }
    public void Calculate()
    {
        _Total = _C + _Cpp + _DotNet;
        _Avg = _Total / 3.0F;
        if (_C < 35 || _Cpp < 35 || _DotNet < 35)
        {
            _Grade = "Fail";
        }
        else
        {
            if (_Avg >= 90)
                _Grade = "Distinction";
            else if (_Avg >= 70)
                _Grade = "First Class";
            else if (_Avg >= 50)
                _Grade = "Second Class";
            else
                _Grade = "Third Class";
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        Marks M = new Marks();
        Console.WriteLine("Enter Marks In Three Subjects");
        M.C = int.Parse(Console.ReadLine());
        M.Cpp = int.Parse(Console.ReadLine());
        M.DotNet = int.Parse(Console.ReadLine());
        M.Calculate();
        Console.WriteLine("Total : {0}", M.Total);
        Console.WriteLine("Aveg  : {0}", M.Avg);
```

```
            Console.WriteLine("Grade : {0}", M.Grade);
        }
    }
}
```

Example : The following example creates a class with the name Salary for accepting basic salary and calculate DA, HRA and gross salary.

```
namespace OOPS
{
    class Salary
    {
        float _Basic, _DA, _HRA, _Gross;
        public float Basic
        {
            get { return _Basic; }
            set { _Basic = value; }
        }
        public float DA
        {
            get { return _DA; }
        }
        public float HRA
        {
            get { return _HRA; }
        }
        public float Gross
        {
            get { return _Gross; }
        }
        public void Calculate()
        {
            if (Basic <= 5000)
            {
                _DA = _Basic * 5 / 100;
                _HRA = _Basic * 10 / 100;
            }
            else if (Basic <= 10000)
            {
                _DA = _Basic * 10 / 100;
                _HRA = _Basic * 15 / 100;
            }
            else if (Basic <= 15000)
            {
                _DA = _Basic * 15 / 100;
                _HRA = _Basic * 20 / 100;
            }
            else
            {
                _DA = _Basic * 20 / 100;
                _HRA = _Basic * 25 / 100;
            }
            _Gross = _Basic + _DA + _HRA;
        }
    }
    class Employees
    {
        static void Main()
        {
```

```
            Salary S = new Salary();
            Console.Write("Enter Basic Salary : ");
            S.Basic = float.Parse(Console.ReadLine());
            S.Calculate();
            Console.WriteLine("Basic Salary : {0}", S.Basic);
            Console.WriteLine("DA : {0}", S.DA);
            Console.WriteLine("HRA : {0}", S.HRA);
            Console.WriteLine("Gross : {0}", S.Gross);
        }
    }
}
```

Static Members Vs Instance Members

The members of a class that can not be accessed without creating an instance for the class are called as instance members and the members of a class that can be accessed without creating an instance and directly by using class name are called as static members. To make a member as static member, declare the member using the keyword **static**. Static members can not accessed by using an instance. While declaring members as static, no need to specify any access modifier. Indexers and destructor can not be static.

Static Fields : When a field is declared as static then that field will exists only one copy for any number of instances of the class. Hence static fields are used to store the data that is to be shared by all instances of the class. For example, if you want to maintain a count of how many instances are created for the class then only alternative is static field.

Static Methods : When a method is declared as static then that method can access only other static members available in the class and it is not possible to access instance members.

Static Classes : You can also create a class it self as static in c#.net 2005. When a class contains every member as static then there is no meaning in creating an instance for the class. In this case to restrict the class from being instantiated, class can be declared as static class.

Constructors : A special method of the class that will be automatically invoked when an instance of the class is created is called as constructor. Constructors are mainly used to initialize private fields of the class while creating an instance for the class. When you are not creating a constructor in the class, then compiler will automatically create a default constructor in the class that initializes all numeric fields in the class to zero and all string and object fields to null. To create a constructor, create a method in the class with same name as class and has the following syntax.

[Access Modifier] ClassName([Parameters])

{

}

Example : The following example creates a class with the name Test with a constructor that initializes fields A and B to 10 and 20.

```
namespace OOPS
{
    class Test
    {
        int A, B;
        public Test()
        {
            A = 10;
            B = 20;
        }


       public void Print()
        {
            Console.WriteLine("A = {0}\tB = {1}", A, B);
        }
    }
    class DefaultConstructor
    {
        static void Main()
        {
            Test T1 = new Test();
            Test T2 = new Test();
            Test T3 = new Test();
            T1.Print();
            T2.Print();
            T3.Print();
        }
    }
}
```


Types of Constructors

Default Constructor : A constructor without any parameters is called as default constructor. Drawback of default constructor is every instance of the class will be initialized to same values and it is not possible to initialize each instance of the class to different values. The above example creates a default constructor and all three instances T1, T2 and T3 are initialized to same values 10 and 20.

Parameterized Constructor : A constructor with at least one parameter is called as parameterized constructor. Advantage of parameterized constructor is you can initialize each instance of the class to different values.

Example : The following example creates a default constructor and a parameterized constructor in the class Test and the instances T1, T2 and T3 are initialized to different values.

```
namespace OOPS
{
    class Test1
    {
        int A, B;
        public Test1()
        {
            A = 10;
            B = 20;
        }
        public void Print()
        {
            Console.WriteLine("A = {0}\tB = {1}", A, B);
        }
        public Test1(int X, int Y)
        {
            A = X;
            B = Y;
        }
    }
    class ParamConstructor
    {
        static void Main()
        {
            Test1 T1 = new Test1();
            Test1 T2 = new Test1(30,40);
            Test1 T3 = new Test1(50,60);
            T1.Print();
            T2.Print();
            T3.Print();
        }
    }
}
```


Copy Constructor : A parameterized constructor that contains a parameter of same class type is called as copy constructor. Main purpose of copy constructor is to initialize new instance to the values of an existing instance.

Example : The following example modifies the above example by creating a copy constructor in the class Test and instance T1 was initialized with default constructor, T2 with Parameterized constructor and T3 with Copy constructor.

```
namespace OOPS
{
    class Test2
    {
        int A, B;
        public Test2()
        {
            A = 10;
            B = 20;
        }
        public void Print()
        {
            Console.WriteLine("A = {0}\tB = {1}", A, B);
        }
        public Test2(int X, int Y)
        {
            A = X;
            B = Y;
        }
        public Test2(Test2 T)
        {
            A = T.A;
            B = T.B;
        }
    }
    class CopyConstructor
    {
        static void Main()
        {
            Test2 T1 = new Test2();
            Test2 T2 = new Test2(30, 40);
            Test2 T3 = new Test2(T2);
            T1.Print();
            T2.Print();
            T3.Print();
        }
    }
}
```


Static Constructor : You can create a constructor as static and when a constructor is created as static, it will be invoked only once for any number of instances of the class and it is during the creation of first instance of the class or the first reference to a static member in the class. Static constructor is used to initialize static fields of the class and to write the code that needs to be executed only once.

Example : The following example creates an instance constructor and a static constructor in the class and 3 instances are created for the class where static constructor is invoked only once. But the instance constructor 3 times once for every instance.

```
namespace OOPS
{
    class Test3
    {
```

```
        public Test3()
        {
            Console.WriteLine("Instance Constructor");
        }
        static Test3()
        {
            Console.WriteLine("Static Constructor");
        }
    }
    class StaticConstructor
    {
        static void Main()
        {
            Test3 T1 = new Test3();
            Test3 T2 = new Test3();
            Test3 T3 = new Test3();
        }
    }
}
```

Private Constructor : You can also create a constructor as private. When a class contains at least one private constructor, then it is not possible to create an instance for the class. Private constructor is used to restrict the class from being instantiated when it contains every member as static.

Some unique points related to constructors are as follows

1. A class can have any number of constructors.
2. A constructor doesn't have any return type even void.
3. A static constructor can not be a parameterized constructor.
4. Within a class you can create only one static constructor.

Destructor : A destructor is a special method of the class that is automatically invoked while an instance of the class is destroyed. Destructor is used to write the code that needs to be executed while an instance is destroyed. To create a destructor, create a method in the class with same name as class preceded with ~ symbol.

Syntax :

```
~ClassName()
{
}
```

Example : The following example creates a class with one constructor and one destructor. An instance is created for the class within a function and that function is called from main. As the instance is created within the function, it will be local to the function and its life time will be expired immediately after execution of the function was completed.

```
namespace OOPS
{
    class Test4
    {
        public Test4()
        {
            Console.WriteLine("An Instance Created");
        }
        ~Test4()
        {
            Console.WriteLine("An Instance Destroyed");
        }
    }
    class Destructor
    {
        public static void Create()
        {
            Test4 T = new Test4();
        }
        static void Main()
        {
            Create();
            GC.Collect();
            Console.Read();
        }
    }
}
```

Object destruction I.e. de allocating the memory of the object is the responsibility of **garbage collector** that is available in CLR of .net framework. Hence an object to be destroyed in .net, you must wait until garbage collector is invoked. Garbage collector will be invoked only in the following two situations.

1. When there is no sufficient memory for new objects.
2. When the application execution comes to an end.

If you want to invoke the garbage collector manual then call the **collect()** method of **GC** class available in .net framework.

Garbage Collector : Garbage Collector is available in CLR of .net framework and it is responsible for complete memory management for the .net applications. Allocating memory for the objects and de allocation of memory when their life time is completed is the responsibility of garbage collector. Various steps performed by garbage collector when it was invoked are as follows.

1. Suspends all the threads in .net application and it will run on a separate thread.
2. It draws a graph of all reachable objects. An object that contains a reference to heap memory area is called as reachable object and the object that contains null is called as unreachable object. All the objects that are not in the graph will be unreachable objects.

3. All unreachable objects with a destructor will be added to a queue called **Freachable queue.**
4. All unreachable objects without a destructor are destroyed and all reachable objects will be moved down the heap to make free memory available at the top of the heap. While moving reachable objects down the heap, garbage collector will update the objects to refer to new memory location.
5. Resumes the threads of the .net application.
6. Destroys all unreachable objects with a destructor available in freachable queue.

Memory Management By Garbage Collector

Garbage collector maintains a pointer with the name **"NextObjPtr"** that initially points to the base address of heap memory area and when memory is allocated for an object, then this pointer will move to next byte of the memory allocated for the object. Hence this pointer always contains the address of a memory location that is to be allocated for next new object created. Prior to allocating memory, it will add required bytes for the new object to address available in **NextObjPtr.** If the address obtained is within the area of heap, then memory is allocated and if the obtained address is beyond the heap memory area then it means that heap memory area is full and it is the time to perform garbage collection.

All the objects in heap memory area up to first garbage collection will be treated as **generation0** and after garbage collection; the new objects created in heap will be treated as **generation1**. Main purpose of maintaining generations is to make the garbage collection fast. The time taken to perform garbage collection of a particular generation will be very less when compared to the garbage collection of entire heap.

If again heap memory becomes full, then garbage collection will be done for **generation0**. If sufficient memory is available after garbage collection on **generation0**, then memory is allocated for new object without performing garbage collection on **generation1**. Otherwise garbage collection will be performed on **generation1** and then memory allocated for the new object. Now onwards the new objects created will be treated as **generation2**.

When heap memory again becomes full, then garbage collection is performed on **generation0** and **generation1** and all remaining reachable objects in **generation0 and generation1** will be treated as **generation0** and all objects in **generation2** will be now moved to **generation1** and newly created objects will be now treated as **generation2** and the same will be continued**.** In .net framework 2.0, garbage collector algorithm contains only three generations.