**Q.1 Explain OOPS (Object Oriented Programming System) Concepts with Example ?**

**Ans:**There are following OOPS Concepts of C# -

1. Class

2. Object

3. Encapsulation

4. Abstraction

5. Inheritance

6. Polymorphism

**1. Class –**

**Class** is a collection of Objects.A class is a blueprint of an object that contains variables for storing data and functions to perform operations on the data.

**Example –**

Public Class Student

{

}

**2. Object –**

**Object** is an instance of a **Class.** An object is a real-world entity that keeps together property states and behaviors.

**Example –**

Public Class Student

{

}

Student objStud=new Student ();

**3. Encapsulation –**

**Encapsulation** is a process of binding the data members and member functions into a single unit.**Encapsulation** provides the ability to hide the internal details of an object from its users. The concept of encapsulation is also known as data hiding or information hiding. In c#, **Encapsulation** is implemented using the [access modifier](http://www.msdotnet.co.in/2012/06/access-specifier-or-modifier-in-c.html) keywords.

**Example –**

Class Demo

    {

        Private int \_mark;

        Public int Mark

        {

            get { return \_mark; }

            set { if (\_mark > 0) \_mark = value; else \_mark = 0; }

        }

    }

TV is encapsulated with a cover and we can operate it with a remote and there is no need to open the TV to change the channel. Here everything is private except the remote, so that anyone can access the remote to operate and change the things in the TV.

**4. Abstraction –**

**Abstraction** is a process of hiding the implementation details and displaying the essential features.**Abstraction**is a mechanism to provide the essential features without describing the background details.

**Example –**

Suppose you have an object Mobile Phone.  
   
Suppose you have 3 mobile phones as in the following:   
   
Nokia 1400 (Features: Calling, SMS)  
Nokia 2700 (Features: Calling, SMS, FM Radio, MP3, Camera)  
Black Berry (Features: Calling, SMS, FM Radio, MP3, Camera, Video Recording, Reading E-mails)  
   
Abstract information (necessary and common information) for the object "Mobile Phone" is that it makes a call to any number and can send SMS.  
   
So that, for a mobile phone object you will have the abstract class as in the following:

   abstract class MobilePhone

    {

        Public void Calling ();

        Public void SendSMS();

    }

    Public class Nokia1400 :MobilePhone

    {

    }

    Public class Nokia2700 :MobilePhone

    {

        Public void FMRadio();

        Public void MP3();

        Public void Camera();

    }

    Public class BlackBerry:MobilePhone

    {

        Public void FMRadio();

        Public void MP3 ();

        Public void Camera ();

        Public void Recording ();

        Public void ReadAndSendEmails();

    }

**5. Inheritance**

**Inheritance** is a process of deriving the new class from already existing class. **Inheritance** is the process of Object Reusability.

**Example–**

Public class ParentClass

    {

        PublicParentClass()

        {

            Console.WriteLine ("Parent Constructor.");

        }

        Public void print()

        {

            Console.WriteLine ("I'm a Parent Class.");

        }

    }

    Public class ChildClass :ParentClass

    {

        PublicChildClass ()

        {

            Console.WriteLine ("Child Constructor.");

        }

        Public static void Main ()

        {

            ChildClass child = new ChildClass();

            child.print();

        }

    }  
   
Output

    Parent Constructor.  
    Child Constructor.  
    I'm a Parent Class.

**6. Polymorphism**

**Polymorphism** means **“Poly”** means**“Many” and “morphism”**means **“forms”.**When a message can be processed in different ways is called **Polymorphism**.

Polymorphism is achieved using following concepts –

1. Method Overloading or Early Binding or Compile Time Polymorphism

2. Method Overriding or Late Binding or Run Time Polymorphism

**Q.3 Explain Method Overloading with example in C#**

**Ans: Method Overloading** is also known as **Early Binding** or**Compile Time Polymorphism**or **Operator Overloading.**

Definition is using the same method name with different type of parameters or different set of parameters is known as **Method Overloading**.

Before going to know all about method overloading, I want to say where it is exactly comes into picture. Lot of programming languages supports default or optional parameters. Example Vb.Net supports optional parameters.

In C#.Net, there is no such type of facility to declare optional parameters as we like.

            Public sub somemethod (a as Integer, b as Integer, optionalc as integer)

            //coding goes here

            End sub

Then this Method overloading concept will be the useful one to overcome this type of problems.

From Framework 4.0 onwards optional parameters are available.

**Method overloading can be achieved by using following things:**

* By changing the number of parameters used.
* By changing the order of parameters.
* By using different data types for the parameters.

**Example -**

using System;

namespace method\_overloading

{

    class Program

    {

        public class Print

        {

            public void display(string name)

            {

                Console.WriteLine ("Your name is : " + name);

            }

            public void display(int age, float marks)

            {

                Console.WriteLine ("Your age is : " + age);

                Console.WriteLine ("Your marks are :" + marks);

            }

        }

        static void Main(string[] args)

        {

            Print obj = new Print ();

            obj.display ("George");

            obj.display (34, 76.50f);

            Console.ReadLine ();

        }

    }

}

**FAQ of Method Overloading**

**Question**-    Can method overloading have same numbers of parameters and Name with different return types?

**Answer**- No, because conflict is happening in methods while changing the parameters.

**Q.1 Explain Access Modifiers in C#.**

**Ans:** Access modifiers are keywords used to specify the declared accessibility of a member or a type. Access modifiers are an integral part of object-oriented programming. They support the concept of encapsulation, which promotes the idea of hiding functionality.

There are following access modifiers in C# -

**1. Public –**

There are no restrictions on accessing public members. Public members are accessible to anywhere in the application or classes.

**Accessibility:**

* Can be accessed by objects of the class
* Can be accessed by derived classes

**2.Private –**

Access is limited to within the class definition. This is the default access modifier type if none is formally specified.

**Accessibility:**

* Cannot be accessed by object
* Cannot be accessed by derived classes.

**3.Protected –**

Access is limited to within the class definition and any class that inherits from the class. A protected member is accessible from within the class in which it is declared, and from within any class derived from the class that declared this member.

**Accessibility:**

* Cannot be accessed by object
* By derived classes

**4.Internal –**

Access is limited exclusively to classes defined within the current project assembly.The **internal** keyword is an access modifier for types and type members. We can declare a class as internal or its member as internal. Internal members are accessible only within files in the same assembly (.dll).

**Accessibility:**

In same assembly (public)

* Can be accessed by objects of the class
* Can be accessed by derived classes

In other assembly (internal)

* Cannot be accessed by object
* Cannot be accessed by derived classes

**5.ProtectedInternal –**

The protected internal accessibility means protected OR internal, not protected AND internal. In other words, a protected internal member is accessible from any class in the same assembly, including derived classes.

**6.DefaultAccess –**

A default access level is used if no access modifier is specified in a member declaration. The following list defines the default access modifier for certain C# types:

**enum**: The default and only access modifier supported is public.

**Class**: The default access for a class is private. It may be explicitly defined using any of the access modifiers.

**Interface**: The default and only access modifier supported is public.

**struct**: The default access is private with public and internal supported as well.

**Q.2 What is Difference between Class and Object?**

**Ans:**

|  |  |  |
| --- | --- | --- |
| SR No | Class | Object |
| 1 | Class is a **blueprint or template** from which objects are created. | Object is an **instance** of a class. |
| 2 | Class is a **group of similar objects**. | Object is a **real world entity** such as pen, laptop, mobile, bed, keyboard, mouse, chair etc. |
| 3 | Class is a **logical** entity. | Object is a **physical** entity. |
| 4 | Class is declared using **class keyword** e.g. class Student{} | Object is created through **new keyword** mainly e.g. Student s1=new Student(); |
| 5 | Class is declared **once**. | Object is created **many times** as per requirement. |
| 6 | Class **doesn't allocated memory when it is created**. | Object **allocates memory when it is created**. |

**Q.3 What is Difference between Abstraction and Encapsulation in C#?**

**Ans:**

|  |  |  |
| --- | --- | --- |
| SR No | Abstraction | Encapsulation |
| 1 | **Abstraction** is a process of hiding the implementation details and displaying the essential features. | **Encapsulation** is a process of binding the data members and member functions into a single unit. |
| 2 | **Abstraction** solves the problem in the design level. | **Encapsulation** solves the problem in the implementation level. |
| 3 | **Abstraction** is used for hiding the unwanted data and giving relevant data. | **Encapsulation** means hiding the code and data into a single unit to protect the data from outside world. |
| 4 | **Abstraction** is set focus on the object instead of how it does it. | **Encapsulation** means hiding the internal details or mechanics of how an object does something. |
| 5 | **Abstraction** is outer layout in terms of design.  For Example: - Outer Look of a iPhone, like it has a display screen. | **Encapsulation** is inner layout in terms of implementation. For Example: - Inner Implementation detail of a iPhone, how Display Screen are connect with each other using circuits |

**Q. Explain Method Overriding with Example in C#.**

**Ans: Method Overriding** is also known as**Late Binding** or**Runtime Time Polymorphism**or **Operator Overriding.**

Creating a method in derived class with same signature as a method in base class is called as **Method Overriding**.Same signature means methods must have same name, same number of arguments and same type of arguments.

**Method Overriding** is possible only in derived classes, but not within the same class.   
When derived class needs a method with same signature as in base class, but wants to execute different code than provided by base class then method overriding will be used.

**Example –**

using System;  
  
namespace methodoverriding  
{  
    class BaseClass  
    {  
        public virtual  string YourCity()  
        {  
            return "New York";  
        }  
    }  
  
    class DerivedClass : BaseClass  
    {  
        public override string YourCity()  
        {  
            return "London";  
        }  
    }  
  
    class Program  
    {  
          
        static void Main(string[] args)  
        {  
            DerivedClassobj = new DerivedClass();  
            string city = obj.YourCity();  
            Console.WriteLine(city);  
            Console.Read();  
        }  
    }  
}

**Output** :  
  
London

**Q.2 what is Difference Method Overloading Class and Method Overriding?**

**Ans:**

|  |  |  |
| --- | --- | --- |
| SR No | Method Overloading | Method Overriding |
| 1 | **Method Overloading** also known as **Early binding**or**Static binding.** | **Method Overriding**also known as **Late binding**or**dynamic binding** |
| 2 | **Method Overloading** also known as **Compile Time Polymorphism.** | **Method Overriding**also known as **Run Time Polymorphism.** |
| 3 | Definition is using the same method name with different type of parameters or different set of parameters, different sequence of parameters is known as **Method Overloading**. | Creating a method in derived class with same signature as a method in base class is called as **Method Overriding**.Same signature means methods must have same name, same number of arguments and same type of arguments. |
| 4 | **Method Overloading** happens in the Same Class. | **Method Overriding** happens in Parent class and derived class. |
| 5 | In **Method Overloading** we don’t want to use **Virtual** and **Override** keyword. | In **Method Overriding**we must have to use **Virtual** and **Override** keyword. |

**Q.2 Explain Virtual Method or Virtual function in C# with Example.**

**Ans:**A **Virtual Method** is a method that can be redefined in derived classes.A **VirtualMethod** is created in the base class that can be overridden in the derived class. We create a **VirtualMethod** in the base class using the “**Virtual”** keyword and that method is overridden in the derived class using the “**override”** keyword. When a method is declared as a **VirtualMethod**in a base class and that method has the same definition in a derived class then there is no need to override it in the derived class. But when a **VirtualMethod**has a different definition in the base class and the derived class then there is a need to override it in the derived class.

**Note**: By default, methods are non-virtual. You cannot override a non-virtual method.

Example –

using System;

class A

{

public virtual void Test()

{

Console.WriteLine("A.Test");

}

}

class B : A

{

public override void Test()

{

Console.WriteLine("B.Test");

}

}

class Program

{

static void Main()

{

// Compile-time type is A.

// Runtime type is A as well.

A ref1 = new A();

ref1.Test();

// Compile-time type is A.

// Runtime type is B.

A ref2 = new B();

ref2.Test();

}

}

**Output** -

A.Test

B.Test

**Q.2 What is Difference between Abstract Class and Interface with Example?**

**Ans:**

|  |  |  |
| --- | --- | --- |
| SR No | Abstract Class | Interface |
| 1 | An **Abstract Class** is a special kind of class that cannot be instantiated. | An **Interface** is not a class. It is an entity that is defined by the word Interface. |
| 2 | A class may inherit only one **AbstractClass**. | A class may inherit several **interfaces**. |
| 3 | Using **Abstract** we cannot achieve **multiple inheritance** | Using an Interface we can achieve **multiple inheritance.** |
| 4 | **Abstract Class** may have **Concrete Methods(Non-Abstract Methods)** | **Interface** may not have **Concrete Methods (Non-Abstract Methods)** they have only **Abstract Methods**. |
| 5 | **Abstract Class** may have Constructors. | **Interface** may not have constructors. |
| 6 | Abstract class we can use any access modifier i.e. public, private, protected, internal etc. | In interface We cannot use any access modifier i.e. public, private, protected, internal etc. because within an interface by default everything is public. |
| 7 | In Abstract Class we can be defined the keywords static, virtual, abstract or sealed. | An Interface member cannot be defined using the keyword static, virtual, abstract or sealed. |

### Q. What is public static void main (String [] args)?

**Ans:**

**Public:**Since the method will be accessed by the other classes (.Net Runtime)."Public" means that main () can be called from anywhere.

**Static**: “static” means that main () doesn't belong to a specific object. .net runtime environment call static method of the class  
where main is defined. By making Main method as static .net  
runtime environment do not need to make object of the class.

**Void**: "void" means that main () returns no value.

**Main**: "main" is the name of a function. Main () is special because it is the start of the program.

**String []**:"String []" means an array of String.

**args**: "args" is the name of the String[] (within the body of main()). "args" is not special; you could name it anything else and the program would work the same.

**String [] args: String** [] args is a single parameter for the method. String [] is the type of the parameter, indicating an array of Strings. args is the name of the parameter. Parameters must be named.

### Q. Why main method is always declared with static?

### Ans: The Main method in C# is always declared with static because it can’t be call in other method of function. The Main method instantiate other objects and variables but there is no any method there that can instantiate main method in C#. On other hand, the main method doesn’t accept parameter from any other function. It only takes parameter as argument via command line argument.

### Q. Explain Constructor in C# with Example.

**Ans**: **Constructor** is a special method of a class which will invoke automatically whenever instance or object of class is created. Constructors are responsible for object initialization and memory allocation of its class. If we create any class without constructor, the compiler will automatically create one default constructor for that class. There is always at least one constructor in every class. Here you need to remember that a class can have any number of constructors and constructors don’t have any return type, not even void and within a class we can create only one static constructor. Generally constructor name should be same as class name. If we want to create constructor in a class we need to create a constructor method name same as class name.

Some of the key points regarding the Constructor are:

* A class can have any number of constructors.
* A constructor doesn't have any return type, not even void.
* A static constructor cannot be a parameterized constructor.
* Within a class you can create only one static constructor.

**Example –**

classSampleA

{

publicSampleA()

{

Console.WriteLine("Sample A Test Method");

}

}

### Q. Explain Types Constructor in C# with Example.

**Ans**:There are following 5 types of constructor’s-

1. Default Constructor

      2.    Parameterized Constructor

      3.    Copy Constructor

      4.    Static Constructor

      5.    Private Constructor

**1. Default Constructor –**

A constructor without having any parameters called default constructor. If we create any class without constructor, the compiler will automatically create one default constructor for that class. The default constructor initializes: 1.All numeric fields in the class to zero.2. All string and object fields to null.

**Example 1–**

|  |
| --- |
| using System;  namespace ConsoleApplication3  {  class Sample  {  public string param1, param2;  public Sample()     // Default Constructor  {  param1 = "Welcome";  param2 = "Aspdotnet-Suresh";  }  }  class Program  {  static void Main(string[] args)  {  Sample obj=new Sample();   // Once object of class created automatically constructor will be called  Console.WriteLine(obj.param1);  Console.WriteLine(obj.param2);  Console.ReadLine();  }  }  } |

When we run above program it will show output like as shown below

**Output -**

|  |
| --- |
| Welcome  Aspdotnet-Suresh |

**Example 2–**

using System;  
namespace DefaultConstractor  
 {  
    class addition  
    {  
        int a, b;

        public addition()   //default contructor

        {

            a = 100;

            b = 175;

        }

        public static void Main()

        {

            additionobj = new addition(); //an object is created , constructor is called

            Console.WriteLine(obj.a);

            Console.WriteLine(obj.b);

            Console.Read();

        }

      }

    }

Now run the application, the output will be as in the following:

**Output–**

100

175

**2. Parameterized Constructors**

A constructor with at least one parameter is called as parameterized constructor. In parameterized constructor we can initialize each instance of the class to different values like as shown below

|  |
| --- |
| using System;  namespace ConsoleApplication3  {  class Sample  {  public string param1, param2;  public Sample(string x, string y)     // Declaring Parameterized constructor with Parameters  {  param1 = x;  param2 = y;  }  }  class Program  {  static void Main(string[] args)  {  Sample obj=new Sample("Welcome","Aspdotnet-Suresh");   // Parameterized Constructor Called  Console.WriteLine(obj.param1 +" to "+ obj.param2);  Console.ReadLine();  }  }  } |

When we run above program it will show output like as shown below

**Output**

|  |
| --- |
| Welcome to Aspdotnet-Suresh |

**Example 2 –**

using System;

namespace Constructor

{  
    class paraconstrctor  
    {  
      public  int a, b;  
      public paraconstrctor(int x, int y)  // decalaringParemetrized Constructor with ingx,y parameter

        {

            a = x;

            b = y;

        }

   }

    class MainClass

    {

        static void Main()

        {

            paraconstrctor v = new paraconstrctor(100, 175);   // Creating object of Parameterized Constructor and ing values

            Console.WriteLine("-----------parameterized constructor example by vithalwadje---------------");

            Console.WriteLine("\t");

            Console.WriteLine("value of a=" + v.a );

            Console.WriteLine("value of b=" + v.b);

            Console.Read();

        }

    }

}

Now run the application, the output will be as in the following:

**Output** –

Value of a=100

Value of b=175

**ConstructorOverloading -**

In c# we can overload constructor by creating another constructor with same method name and different parameters like as shown below

|  |
| --- |
| using System;  namespace ConsoleApplication3  {  class Sample  {  public string param1, param2;  public Sample()     // Default Constructor  {  param1 = "Hi";  param2 = "I am Default Constructor";  }  public Sample(string x, string y)     // Declaring Parameterized constructor with Parameters  {  param1 = x;  param2 = y;  }  }  class Program  {  static void Main(string[] args)  {  Sample obj = new Sample();   // Default Constructor will Called  Sample obj1=new Sample("Welcome","Aspdotnet-Suresh");   // Parameterized Constructor will Called  Console.WriteLine(obj.param1 + ", "+obj.param2);  Console.WriteLine(obj1.param1 +" to " + obj1.param2);  Console.ReadLine();  }  } |

When we run above program it will show output like as shown below

Output

|  |
| --- |
| Hi, I am Default Constructor  Welcome to Aspdotnet-Suresh |

**3. Copy Constructor**

A parameterized constructor that contains a parameter of same class type is called as **Copyconstructor**. The constructor which creates an object by copying variables from another object is called a **copyconstructor**. The purpose of a **copyconstructor** is to initialize a new instance to the values of an existing instance. Main purpose of **copyconstructor** is to initialize new instance to the values of an existing instance. Check below example for this

**Example -**

|  |
| --- |
| using System;  namespace ConsoleApplication3  {  class Sample  {  public string param1, param2;  public Sample(string x, string y)  {  param1 = x;  param2 = y;  }  public Sample(Sample obj)     // Copy Constructor  {  param1 = obj.param1;  param2 = obj.param2;  }  }  class Program  {  static void Main(string[] args)  {  Sample obj = new Sample("Welcome", "Aspdotnet-Suresh");  // Create instance to class Sample  Sample obj1=new Sample(obj); // Here obj details will copied to obj1  Console.WriteLine(obj1.param1 +" to " + obj1.param2);  Console.ReadLine();  }  }  } |

When we run above program it will show output like as shown below

**Output**

|  |
| --- |
| Welcome to Aspdotnet-Suresh |

**Example 2 –**

using System;

namespace copyConstractor

{

    class employee

    {

        private string name;

        privateint age;

        public employee(employee emp)   // declaring Copy constructor.

        {

            name = emp.name;

            age = emp.age;

        }

        public employee(string name, int age)  // Instance constructor.

        {

            this.name = name;

            this.age = age;

        }

        public string Details     // Get deatils of employee

        {

            get

            {

                return  " The age of " + name +" is "+ age.ToString();

            }

        }

    }

    class empdetail

    {

        static void Main()

        {

            employee emp1 = new employee("Vithal", 23);  // Create a new employee object.

            employee emp2 = new employee(emp1);          // here is emp1 details is copied to emp2.

            Console.WriteLine(emp2.Details);

            Console.ReadLine();

        }

    }

}

Now run the program, the output will be as follows:

**Output –**

Age of Vitthal is 23

**4. Static Constructor -**

When we declared constructor as static it will be invoked only once for any number of instances of the class and it’s 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.

|  |
| --- |
| using System;  namespace ConsoleApplication3  {  class Sample  {  public string param1, param2;  static Sample()  {  Console.WriteLine("Static Constructor");  }  public Sample()  {  param1 = "Sample";  param2 = "Instance Constructor";  }  }  class Program  {  static void Main(string[] args)  {  // Here Both Static and instance constructors are invoked for first instance  Sample obj=new Sample();  Console.WriteLine(obj.param1 + " " + obj.param2);  // Here only instance constructor will be invoked  Sample obj1 = new Sample();  Console.WriteLine(obj1.param1 +" " + obj1.param2);  Console.ReadLine();  }  }  } |

When we run above program we will get output like as shown below

**Output**

|  |
| --- |
| Static Constructor  Sample Instance Constructor  Sample Instance Constructor |

**Importance points of static constructor**

-      Static constructor will not accept any parameters because it is automatically called by CLR.

-      Static constructor will not have any access modifiers.

-      Static constructor will execute automatically whenever we create first instance of class

-      Only one static constructor will allowed.

**5. Private Constructor**

When a constructor is created with a private specifier, it is not possible for other classes to derive from this class, neither is it possible to create an instance of this class. They are usually used in classes that contain static membersonly.

Private constructor is a special instance constructor used in a class that contains static member only. If a class has one or more private constructor and no public constructor then other classes is not allowed to create instance of this class this mean we can neither create the object of the class nor it can be inherit by other class. The main purpose of creating private constructor is used to restrict the class from being instantiated when it contains every member as static.

|  |
| --- |
| using System;  namespace ConsoleApplication3  {  public class Sample  {  public string param1, param2;  public Sample(string a,string b)  {  param1 = a;  param2 = b;  }  private Sample()  // Private Constructor Declaration  {  Console.WriteLine("Private Constructor with no prameters");  }  }  class Program  {  static void Main(string[] args)  {  // Here we don't have chance to create instace for private constructor  Sample obj = new Sample("Welcome","toAspdotnet-Suresh");  Console.WriteLine(obj.param1 +" " + obj.param2);  Console.ReadLine();  }  }  } |

**Output**

|  |
| --- |
| Welcome to Aspdotnet-Suresh |

In above method we can create object of class with parameters will work fine. If create object of class without parameters it will not allow us create.

|  |
| --- |
| // it will works fine  Sample obj = new Sample("Welcome","toAspdotnet-Suresh");  // it will not work because of inaccessability  Sample obj=new Sample(); |

**Important points of private constructor**

-      One use of private construct is when we have only static member.

-      Once we provide a constructor that is either private or public or any, the compiler will not allow us to add public constructor without parameters to the class.

-      If we want to create object of class even if we have private constructors then we need to have public constructor along with private constructor

**Q. Explain difference between Value Type and Reference Type in C#.**

**Ans:**

|  |  |  |
| --- | --- | --- |
| SR No | Value Type | Reference Type |
| 1 | Variables that store data are called **Value Types**. | Variables that store reference to actual data are called **Referencetypes.** |
| 2 | **ValueTypes** are stored on **StackMemory**. | **ReferenceType** are stored on **HeapMemory** |
| 3 | Memory is allocated at compile time | Memory is allocated at run time |
| 4 | Cannot contain null values. However this can be achieved by nullable types | Can contain null values. |
| 5 | When we assign a value type to another value type, a field-by-field copy is made. | When we copy a reference type to another reference type, only the memory address is copied. |
|  | **Value types** cannot be inherited | **Reference types** can be inherited |
| 6 | **Examples** – int,float,enum,struct | **Examples –** Class,string,object |

**Q. Explain Destructors in C# with Example.**

**Ans:** A **Destructor** is a function with the same name as the name of the class but starting with the character ~ .Dot Net will clean up the un-used objects by using garbage collection process. It internally uses the destruction method to clean up the un-used objects.**Destructors** are invoked automatically and can't be invoked explicitly.**Destructor** can't have any modifiers like private, public etc. If we declare a destructor with a modifier, the compiler will show an error. Also destructor will come in only one form, without any arguments. There is no parameterized destructor in C#. **Destructors** are invoked automatically and can't be invoked explicitly. An object becomes eligible for garbage collection, when it is no longer used by the active part of the program.

**Syntax**:  
  
~<ClassName>  
{}  
  
**Example**:  
  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;

namespaceBRK.ConstructorExample  
{  
    class Welcome  
    {  
        // Default constructor  
        public Welcome()  
        {  
            Console.WriteLine("Welcome message from Default Constructor...");  
        }

        // Destructor  
        ~Welcome()  
        {  
            Console.WriteLine("Destructor called");          
        }  
    }  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            // Creating object for Welcome class  
            // This will called default constructor  
            Welcome obj = new Welcome();

            Console.Read();  
        }  
    }  
}  
  
**Note**: Destructor will call after execution of the program.

**Q. Explain difference Struct and Class in C#.**

**Ans:**

|  |  |  |
| --- | --- | --- |
| SR No | Struct | Class |
| 1 | **Struct** is Value Type in C#. | **Class** is a reference type in C#. |
| 2 | **Struct**inheritfrom**System.ValueType**namespace in C#. | **Class** inheritfrom**System.Object**namespace in C#. |
| 3 | The **struct** value will be stored on the **stackmemory**. | **Class** object will be stored on the **Heapmemory**. |
| 4 | **Struct**can have only**constructor.** | **Class** may have**Constructor**and **Destructor.** |
| 5 | The struct can't have the default constructor | The **class** will have the default constructor. |
| 6 | The struct can't use the protected or protected internal modifier. | The class can use all the access modifiers. |
| 7 | Structs are usually used for smaller amounts of data. | Classes are usually used for large amounts of data. |
| 8 | Structs cannot be inherited. | Classes can be inherited. |
| 9 | A structure can't be abstract. | Classes can be abstract. |

**Q. Explain difference String and StringBuilder in C#.**

**Ans:**

|  |  |  |
| --- | --- | --- |
| SR No | String | StringBuilder |
| 1 | **String** inheritsfrom**Sytem.String**namespace**.** | **StringBuilder** inheritsfrom**Sytem.Text**namespace**.** |
| 2 | **String** is **immutable**. Immutable means once we create string object we cannot modify. | **StringBuilder** is **mutable**. Immutable means once we create string object we can modify it. |
| 3 | Performance wise **string** is slow because every time it will create new instance | Performance wise **stringbuilder** is high because it will use same instance of object to perform any action |
| 4 | In string we don’t have append keyword | In **StringBuilder** we can use append keyword |

**Example of String –**

String is immutable. Immutable means once we create string object we cannot modify. Any operation like insert, replace or append happened to change string simply it will discard the old value and it will create new instance in memory to hold the new value.

|  |
| --- |
| string str = "hi";  // create a new string instance instead of changing the old one  str += "test";  str += "help"; |

**Example of StringBuilder**–

String builder is mutable it means once we create string builder object we can perform any operation like insert, replace or append without creating new instance for every time.

|  |
| --- |
| StringBuildersb = new StringBuilder("");  sb.Append("hi");  sb.Append("test ");  string str = sb.ToString(); |

**Q.2 what is Difference between Array and ArrayList?**

**Ans:**

|  |  |  |
| --- | --- | --- |
| SR No | Array | ArrayList |
| 1 | **Arrays** are strongly typed collection of same datatype. | **ArrayList**are not strongly type collection.  It will store values of different datatypes or same datatype. |
| 2 | **Arrays** are fixed length that cannot be changed during runtime. | **ArrayList**size will increase or decrease dynamically it can take any size of values from any data type. |
| 3 | **Arrays** belong to **System.Array** namespace. | **Arraylist** belongs to **System.Collection** namespaces. |
| 4 | **Array** is a reference type data type | **ArrayList**is a class . |

**Declaration of Arrays**

Generally we will declare arrays with fixed length and store values like as shown below

|  |
| --- |
| string[] arr=new string[2];  arr[0] = "welcome";  arr[1] = "Aspdotnet-suresh"; |

In above code I declared array size 2 that means we can store only 2 string values in array.

**Declaration of Arraylist -**

To know how to declare and store values in array lists check below code

|  |
| --- |
| ArrayList strarr = new ArrayList();  strarr.Add("welcome"); // Add string values  strarr.Add(10);   // Add integer values  strarr.Add(10.05); // Add float values |

If you observe above code I haven’t mentioned any size in array list we can add all the required data there is no size limit and we will use add method to bind values to array list.

**Q.2 what is Difference between ArrayList and HashTable?**

**Ans:**

|  |  |  |
| --- | --- | --- |
| SR No | ArrayList | HashTable |
| 1 | Data in**Arraylist**is stored in as only the **value**. | Data in **Hashtable** is stored in the **key-value** pair. |
| 2 | **ArrayList** is a List. | **HashTable** is a map. |
| 3 | To access value from **ArrayList**, you need to pass **Index.** | To access value from **HashTable**, you need to pass **name** |
| 4 | In this we can only add items to the list | Here we can add data with the key |
| 5 | Anything stored in **Arraylist**is stored as object | The collection class that supports unique key is **HashTable.** |
| 6 | Searching is Slower as compared to **ArrayList.** | Retrieving by key in **Hashtable** is faster than retrieving in  **Arraylist.** |
| 7 | In**Arraylist**you can store similar kind of data. | In **HashTable** you can store different types of data asvalues are stored as objects internally. |
| 8 | **Arraylist**can be sorted using the sort method. | **HashTable**cannot be sorted. |

**Q.2 what is Difference between HashTable and Dictionary?**

**Ans:**

|  |  |  |
| --- | --- | --- |
| SR No | HashTable | Dictionary |
| 1 | **HashTable** is a Non-Generic type | **Dictionary** is a generic type |
| 2 | **HashTable**values need to be boxed or unboxed because it stored the values and keys as objects. | **Dictionary** is type’s means that the values need not to boxing. |
| 3 | When you try to get the value of key which does not existshashtable returns null value. | When you try to get the value of key which does not exists in the collection, the dictionary throws an exception of 'KeyNotFoundException' |
| 4 | When we retrieve the record in collection the **HashTable**does not maintain the order of entries. | When we retrieve the record in collection **dictionary** maintains the order of entries by which entries were added. |
| 5 | **Hashtable** relies on rehashing. | **Dictionary** relies on chaining |
| 6 | In **Hashtable** boxing/unboxing (value types need boxing) will happened and which may have memory consumption as well as performance penalties. | While perform operations **Dictionary** is faster because there is no boxing/unboxing (value types don't need boxing) |
| 7 | The **Hashtable** is a weakly typed data structure, so you can add keys and values of any Object Type to the **Hashtable.** | The **Dictionary** class is a strongly types < T Key, T Value > and you must specify the data types for both the key and value. |
| 8 | **Declaration** –  **Hashtable** numbers = new **Hashtable**(); | **Declaration** –  Dictionary<int, string> dictionary = new Dictionary<int, string >(); |

**Example of HashTable and Dictionary –**

Collections & Generics are useful for handling group of Objects. In .net, all the collections objects comes under the interface IEnumerable, Which in turn has ArrayList(Index-Value)) & HashTable(Key-Value). After .net framework 2.0, ArrayList & HashTable were replaced with List & Dictionary. Now, the Arraylist & HashTable are no more used in now a days projects.

Coming to difference between HashTable & Dictionary, Dictionary is generic where as Hastable is not Generic. We can add any type of object to HashTable, but while retrieving we need to Cast it to the required Type. So, it is not type safe. But to dictionary, while declaring itself we can specify the type of Key & Value, so no need to cast while retrieving.

Let's look at an example:

**HashTable**

classHashTableProgram

{

static void Main(string[] args)

{

Hashtableht = new Hashtable();

ht.Add(1, "One");

ht.Add(2, "Two");

ht.Add(3, "Three");

foreach (DictionaryEntry de in ht)

{

int Key = (int)de.Key; //Casting

string value = de.Value.ToString(); //Casting

Console.WriteLine(Key + " " + value);

}

}

}

**Dictionary-**

classDictionaryProgram

{

static void Main(string[] args)

{

Dictionary<int, string>dt = new Dictionary<int, string>();

dt.Add(1, "One");

dt.Add(2, "Two");

dt.Add(3, "Three");

foreach (KeyValuePair<int, String>kv in dt)

{

Console.WriteLine(kv.Key + " " + kv.Value);

}

}

}

**Q.2 Explain Constant, ReadOnlywith Example.**

**Ans :**

**Const(Constant)-**

1.    **Const** can only be initialized at the time of declaration of the field.

      2.    **Const** values will evaluate at compile time only.

      3.    **Const** value can’t be changed these will be same at all the time.

      4.    This type of fields are required when one of the field values remains constant throughout the system like Pi will remain same in your Maths Class.

5. By default constant are static, hence you cannot define a constant type as static.

**ReadOnly–**

1.    The value will be initialized either declaration time or the constructor of the class allowing you to pass the value at run time.

      2.    Read only values will evaluate at runtime only.

**Example –**

public class Const\_VS\_Readonly

{

public const int I\_CONST\_VALUE = 2;

public readonly int I\_RO\_VALUE;

public Const\_VS\_Readonly()

{

I\_RO\_VALUE = 3;

}

}

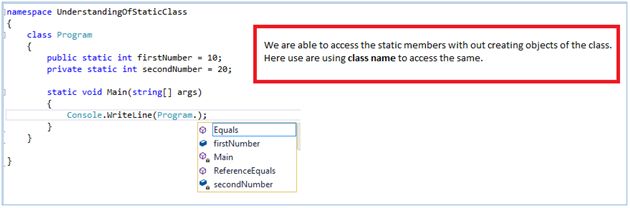
**Q.2 Explain Static with Example in C#.**

**Ans :**

**Static Variable –**

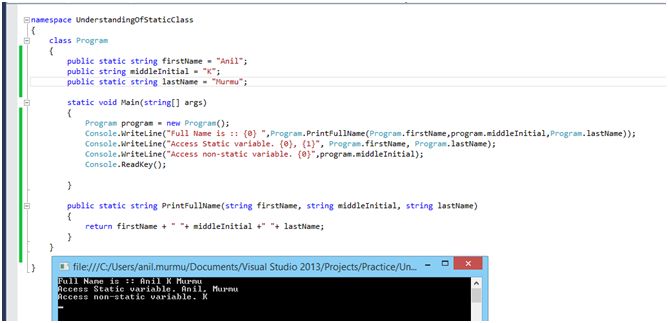
A static variable shares the value of it among all instances of the class.

When we don’t want to create multiple instance of a variable, i.e., we want to access the same value across multiple instance of class, then we can opt for a static variable.  
To access static variable we don’t need to create an object of the class. It can be accessible by the class name.



**Static Method –**

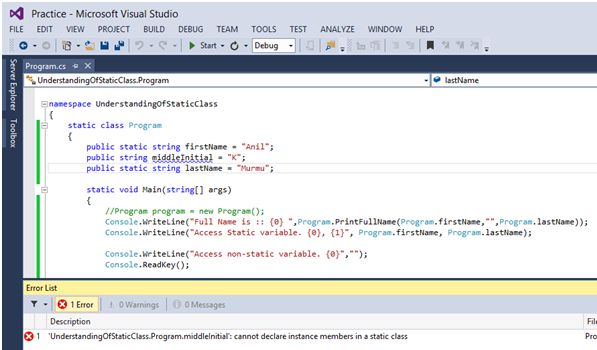
When we have a scenario which focuses more into the output only without any requirement to store or retrieve data that is unique to any specific instance of a class, then we can opt for Static Methods.   
To access the static methods also we need to use the class name. No instance is required.



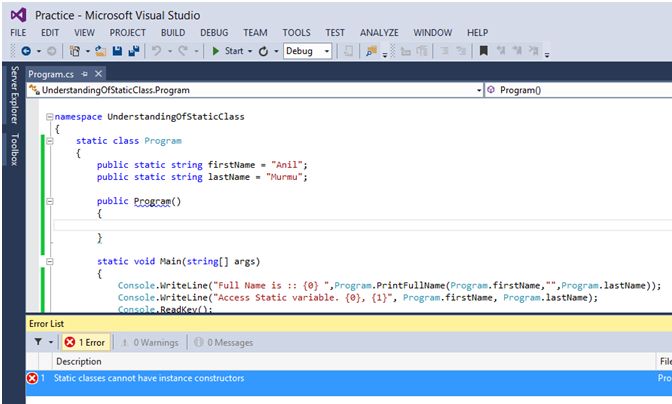
**Question:** Can static method access non-static fields?  
**Answer**   
The Answer is **yes**. Static methods can directly access static fields without using class name also. However, to access non-static fields an object needs to be created. An example has been shown in the above image.

**StaticClass -**

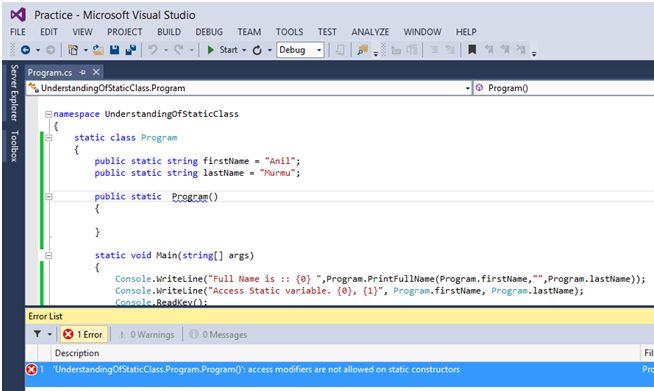
When we have all member functions and data member in a class declared as static we should declare the class as Static. So, in other words, we can say a static class should not contain any non-static fields/methods.  
Let’s see in an example.



In the above example we are getting a compile time Error. Saying cannot declare instance members in a static class.  
**Question:** If in a static class, we don’t have non static members/functions then does it hold any default constructor?  
**Answer** -Yes, it does hold a constructor but it is also declared as static.



In the above example we are trying to create an instance constructor but during compile time we are getting an error. So let’s declare it as static.  
Now we have declared the constructor as static but again at compile time we are getting one more error



So, let's correct the error. To correct the error, we have to remove the public access modifier.   
**Question:** Can static constructor have public/private/internal/protected internal access modifiers?  
**Answer** No, static constructor doesn’t contain any access modifiers. We have already seen the implementation in previous example.

**Q.2 Explain difference between ref and out keyword in C#.**

**Ans:**

Both **ref** and **out** parameters are used to pass arguments within a method. These **ref** and **out** parameters are useful whenever your method wants to return more than one value.

**Ref Keyword–**

If you want to pass a variable as **ref** parameter you need to initialize it before you pass it as ref parameter to method. The **ref** keyword is used to pass an argument as a reference. This means that when value of that parameter is changed in the method, it gets reflected in the calling method.

**Example code**

public static string GetNextName(ref int id)  
{  
    string returnText = "Next-" + id.ToString();  
    id += 1;  
    return returnText;  
}  
static void Main(string[] args)  
{  
    int i = 1;  
    Console.WriteLine("Previous value of integer i:" + i.ToString());  
    string test = GetNextName(ref i);  
    Console.WriteLine("Current value of integer i:" + i.ToString());  
}

**Output**

Ref Output

**Out Keyword–**

If you want to pass a variable as out parameter you don’t need to initialize it before you pass it as out parameter to method. Out keyword also will pass parameter as a reference but here out parameter must be initialized in called method before it return value to calling method.

**Example** –

class Program

{

static void Main()

{

int i,j; // No need to initialize variable

Outsample(out i, out j);

Console.WriteLine(i);

Console.WriteLine(j);

}

public static int Outsample(out int val1, out int val2)

{

val1 = 5;

val2 = 10;

return 0;

}

}

**Output** –

**5**

**10**

|  |  |  |
| --- | --- | --- |
| SR No | ref | out |
| 1 | The parameter or argument must be initialized first before it is passed to ref. | It is not compulsory to initialize a parameter or argument before it is passed to an out. |
| 2 | It is not required to assign or initialize the value of a parameter (which is passed by ref) before returning to the calling method. | A called method is required to assign or initialize a value of a parameter (which is passed to an out) before returning to the calling method. |
| 3 | Passing a parameter value by Ref is useful when the called method is also needed to modify the pass parameter | Declaring a parameter to an out method is useful when multiple values need to be returned from a function or method. |
| 4 | It is not compulsory to initialize a parameter value before using it in a calling method. | A parameter value must be initialized within the calling method before its use. |
| 5 | When we use REF, data can be passed bi-directionally. | When we use OUT data is passed only in a unidirectional way (from the called method to the caller method). |

**Q. Explain Garbage Collection in .NET with Example.**

**Ans:**

The **Garbage collection** is very important technique in the .Net framework to free the unused managed code objects in the memory and free the space to the process.CLR has garbage collector that executes as a part of our program and responsible for reclaiming the memory of no longer used objects. Garbage collector frees the memory for objects that are no longer referenced and keeps the memory for future allocations.

**Generations of Garbage Collection –**

There are three generations in garbage collection generation 0,1,2.Geneartion 0 are fresh created objects, generation 1 are objects which are bit older than generation 0 and generation 2 are oldest objects.

1. **Generation 0**

This is the youngest generation and contains the newly created objects. Generation 0 has short-lived objects and collected frequently. The objects that survive the Generation 0 are promoted to Generation 1.

**Example**: A temporary object.

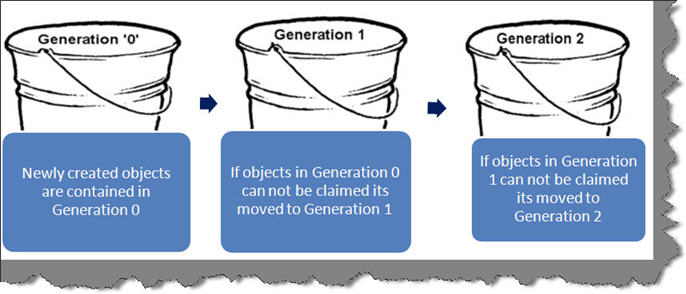
1. **Generation 1**

This generation contains the longer lived objects that are promoted from generation 0. The objects that survive the Generation 1 are promoted to Generation 2. Basically this generation serves as a buffer between short-lived objects and longest-lived objects.

1. **Generation 2**

This generation contains the longest lived objects that are promoted from generation 1 and collected infrequently.

**Example**: An object at application level that contains static data which is available for the duration of the process.



**Q. Explain difference between finalize and dispose method with example.**

**Ans:**

**Dispose () Method -**

     -  This dispose method will be used to free unmanaged resources like files, database connection etc.

     -  To clear unmanaged resources we need to write code manually to raise **dispose ()** method.

     -  This **Dispose ()** method belongs to **IDisposable** interface.

     -  If we need to implement this method for any custom classes we need to inherit the class from **IDisposable** interface.

     -  It will not show any effect on performance of website and we can use this method whenever we want to free objects immediately.

**Example –**

public class TestDispose : IDisposable

{

private bool disposed = false;

public void Dispose()

{

Dispose(true);

}

protected virtual void Dispose(bool disposing)

{

if (!disposed)

{

if (disposing)

{

// clean unmanged objects

}

// clean unmanaged objects).

disposed = true;

}

}

}

**Finalize() Method-**

     -  This method also free unmanaged resources like database connections, files etc.…

     -  It is automatically raised by garbage collection mechanism whenever the object goes out of scope.

     -  This method belongs to object class.

     -  We need to implement this method whenever we have unmanaged resources in our code and make sure these resources will be freed when garbage collection process done.

     -  It will show effect on performance of website and it will not suitable to free objects immediately.

**Example -**

// Implementing Finalize method

public class Sample

{

//At runtime destructor automatically Converted to Finalize method.

~Sample()

{

// your clean up code

}

}

|  |  |  |
| --- | --- | --- |
| SR No | Dispose | Finalize |
| 1 | It is used to free unmanaged resources like files, database connections etc. at any time. | It can be used to free unmanaged resources (when you implement it) like files, database connections etc. held by an object before that object is destroyed. |
| 2 | Explicitly, it is called by user code and the class which is implementing dispose method, must has to implement IDisposable interface. | Internally, it is called by Garbage Collector and cannot be called by user code. |
| 3 | It belongs to IDisposable interface. | It belongs to Object class. |
| 4 | It's implemented by implementing IDisposable interface Dispose () method. | It's implemented with the help of destructor in C++ & C#. |
| 5 | There is no performance costs associated with Dispose method. | There is performance costs associated with Finalize method since it doesn't clean the memory immediately and called by GC automatically |

**Q. What’s the difference between the System.Array.CopyTo () and System.Array.Clone () with Example in C#?**

**Ans:**

|  |  |  |
| --- | --- | --- |
| SR No | System.Array.CopyTo() | System.Array.Clone() |
| 1 | Array.CopyTo copies the elements of one array to another pre-existing array starting from a given index (usually 0). | Array.Clone creates a copy of an array as an object. It therefore needs to be cast to the actual array type before it can be used to do very much. |
| 2 | In this case Both arrays must be single dimensional. | It works on both single and multi-dimensional arrays. |
| 3 | System.Array.CopyTo () method performs the deep copy means it will create new instance of each element object. | System.Array.Clone() method performs shallow copy i.e. shallow copy contains reference of same object. |
| 4 | System.Array.CopyTo () require having a destination array. | System.Array.Clone ()  the destination array need not exist yet since a new one is created. |
| 5 | It requires a destination array. | It does not require destination array. |

**Q. Explain difference between throw and throw ex in C#.**

**Ans:**

**Throw -**

In Throw, the original exception stack trace will be retained. To keep the original stack trace information, the correct syntax is 'throw' without specifying an exception. In other word Throw basically retains the stack information and adds to the stack information in the exception that it is thrown.

**Declaration of throw**

try

{

// do some operation that can fail

}

catch (Exception ex)

{

// do some local cleanup

throw;

}

**Throw ex –**

In Throw ex, the original stack trace information will get override and you will lose the original exception stack trace. I.e. 'throw ex' resets the stack trace and this makes it hard to find the original code line number that has thrown the exception.

**Declaration of throw ex**

try

{

// do some operation that can fail

}

catch (Exception ex)

{

// do some local cleanup

throw ex;

}

}

**Q. Explain difference between IS and AS operator in C#.**

**Ans:**

### IS operator -

[IS operator](https://msdn.microsoft.com/en-us/library/scekt9xw.aspx) checks if an object can be cast to a specific type. The IS operator checks whether the type of a given object is compatible with the new object type. It returns Boolean type value:**true** if given object is compatible with new one, else **false**. In this way IS operator help you to do safe type casting.

**Example:**

if (obj is string)

{

...

}

### AS operator -

The [**AS**operator](https://msdn.microsoft.com/en-us/library/cscsdfbt.aspx) attempts to cast an object to a specific type, and returns null if it fails. The**AS** operator also checks whether the type of a given object is compatible with the new object type. It returns non-null if given object is compatible with new one, else null. In this way AS operator help you to do safe type casting. The above code can be re-written by using AS operator in a better way.

**Example:**

stringstr = obj as string;

if(str != null)

{

...

}

**Q. Explain difference between Boxing and Unboxing in C#.**

**Ans:**

**Boxing –**

Converting Value type to reference type is called **Boxing**.Implicit conversion of a value type (int, char etc.) to a reference type (object), is known as Boxing. In boxing process, a value type is being allocated on the heap rather than the stack.

**Example-**

int Val = 1;

Object Obj = Val; //Boxing

**UnBoxing –**

Converting reference type to value type using casting is called **Unboxing**.Explicit conversion of same reference type (which is being created by boxing process); back to a value type is known as unboxing. In unboxing process, boxed value type is unboxed from the heap and assigned to a value type which is being allocated on the stack.

**Example-**

int Val = 1;

Object Obj = Val; //Boxing

inti = (int)Obj; //Unboxing

**Q. Explain difference between break and continue in C#.**

**Ans:**

 Using **break** statement, you can 'jump out of a loop' whereas by using **continue** statement, you can 'jump over one iteration' and then resume your loop execution.

**Example of break Statement -**

for (inti = 0; i < 10; i++) {

if (i == 2) {

break;

}

Console.WriteLine(“Number is”+i);

 }

**Output –**

Number is 0

Number is 1

**Example of Continue Statement -**

for (inti = 0; i <5; i++) {

if (i == 2) {

break;

}

Console.Write(“Number is”+i);

 }

**Output –**

Number is 0Number is 1 Number is 3 Number is 4 Number is 5

**Q. Explain Delegate in C# with example.**

**Ans:**

C# **delegates** are similar to pointers to functions, in C or C++.All **delegates** are implicitly derived from the **System.Delegate** class. A **delegate** is a reference type variable that holds the reference to a method. The reference can be changed at runtime.**Delegates** are especially used for implementing events and the call-back methods. Whenever we want to create delegate methods we need to declare with delegate keyword and delegate methods signature should match exactly with the methods which we are going to hold like same return types and same parameters otherwise **delegate** functionality won’t work if signature not match with methods.

**What is the use of Delegates?**

Suppose if you have multiple methods with same signature (return type & number of parameters) and want to call all the methods with single object then we can go for delegates.

**Delegates are two types**

      -   Single Cast Delegates

      -  Multi Cast Delegates

**Example –**

public delegate int DelegatSample(int a,int b);

public class Sampleclass

{

public int Add(int x, int y)

{

return x + y;

}

public int Sub(int x, int y)

{

return x - y;

}

}

class Program

{

static void Main(string[] args)

{

Sampleclass sc=new Sampleclass();

DelegatSample delgate1 = sc.Add;

int i = delgate1(10, 20);

Console.WriteLine(i);

DelegatSample delgate2 = sc.Sub;

int j = delgate2(20, 10);

Console.WriteLine(j);

}

}

**Output –**

Add Result: 30

Sub Result: 10

**Q. Explain Types of Delegate with Example in C#.**

**Ans:**

**1 .Single Cast Delegate –**

**Single**-**cast** delegate means which hold address of single method like as explained in above example. A **Single-cast** derives from the **System.Delegate** class. It contains reference to one method only at a time.

**2 .Multicast Delegates –**

**Multicast** delegate is used to hold address of multiple methods in single delegate. To hold multiple addresses with delegate we will use overloaded **+=** operator and if you want remove addresses from delegate we need to use overloaded operator **-=**.

**Multicast** delegates will work only for the methods which have return type only void. If we want to create a multicast delegate with return type we will get the return type of last method in the invocation listAmulticast delegate derives from the **System.MulticastDelegate** class. It contains an invocation list of multiple methods.

**Example-**

public delegate void MultiDelegate(int a,int b);

public class Sampleclass

{

public static void Add(int x, int y)

{

Console.WriteLine("Addition Value: "+(x + y));

}

public static void Sub(int x, int y)

{

Console.WriteLine("Subtraction Value: " + (x - y));

}

public static void Mul(int x, int y)

{

Console.WriteLine("Multiply Value: " + (x \* y));

}

}

class Program

{

static void Main(string[] args)

{

Sampleclass sc=new Sampleclass();

MultiDelegatedel = Sampleclass.Add;

del += Sampleclass.Sub;

del += Sampleclass.Mul;

del(10, 5);

Console.ReadLine();

}

}