C# -NOTES

**OOPS Concepts**

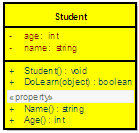
**Class:**

It is a collection of objects.

**Object:**

It is a real time entity.

An object can be considered a "**thing**" that can perform a set of **related**activities. The set of activities that the object performs defines the object's behavior. For example, the hand can grip something or a**Student**(object) can give the name or address. In pure **OOP**terms an object is an instance of a class

[](http://3.bp.blogspot.com/-jk1xzP_jph0/UG3cM0KD95I/AAAAAAAAB5Y/aBkA6f1KBDk/s1600/Student.png)

The above template describe about object **Student**

Class is composed of three things name, attributes, and operations

public class student

{

}

student objstudent=new student ();

According to the above sample we can say that **Student**object, named **objstudent,**has created out of the student class.

In real world you will often find many individual objects all of the same kind. As an example, there may be thousands of other bicycles in existence, all of the same make and model. Each bicycle has built from the same blueprint. In object-oriented terms, we say that the bicycle is an instance of the class of objects known as bicycles. In the software world, though you may not have realized it, you have already used classes. For example, the **Textbox**control, you always used, is made out of the **Textbox**class, which defines its appearance and capabilities. Each time you drag a **Textbox**control, you are actually creating a new instance of the **Textbox**class.

**Encapsulation**:

Encapsulation is a process of binding the data members and member functions into a single unit.

**Example** for encapsulation is **class**. A class can contain data structures and methods.

Consider the following class

public class Aperture

{

public Aperture ()

{

}

protected double height;

protected double width;

protected double thickness;

public double get volume()

{

Double volume=height \* width \* thickness;

if (volume<0)

return 0;

return volume;

}

}

In this example we encapsulate some data such as height, width, thickness and method Get Volume. Other methods or objects can interact with this object through methods that have public access modifier

**Abstraction:**

Abstraction is a process of hiding the implementation details and displaying the essential features.

**Example1**: A Laptop consists of many things such as processor, motherboard, RAM, keyboard, LCD screen, wireless antenna, web camera, usb ports, battery, speakers etc. To use it, you don't need to know how internally LCD screens, keyboard, web camera, battery, wireless antenna, speaker’s works.  You just need to know how to operate the laptop by switching it on. Think about if you would have to call to the engineer who knows all internal details of the laptop before operating it. This would have highly expensive as well as not easy to use everywhere by everyone.

So here the Laptop is an object that is designed to hide its complexity.

How to abstract: - By using **Access Specifiers**

**.Net has five access Specifiers**

**Public** -- Accessible outside the class through object reference.

**Private** -- Accessible inside the class only through member functions.

**Protected** -- Just like private but Accessible in derived classes also through member

functions.

**Internal**-- Visible inside the assembly. Accessible through objects.

**Protected Internal** -- Visible inside the assembly through objects and in derived classes outside the assembly through member functions.

Let’s try to understand by a practical example:-

public class Class1

    {

        int  i;                                         //No Access specifier means private

        public  int j;                                        // Public

        protected int k;                             //Protected data

        internal int m;                        // Internal means visible inside assembly

        protected internal int n;                   //inside assembly as well as to derived classes outside assembly

        static int x;                                 // This is also private

        public static int y;                       //Static means shared across objects

        [DllImport("MyDll.dll")]

        public static extern int MyFoo();       //extern means declared in this assembly defined in some other assembly

        public void myFoo2()

        {

            //Within a class if you create an object of same class then you can access all data members through object reference even private data too

            Class1 obj = new Class1();

            obj.i =10;   //Error can’t access private data through object.But here it is accessible.:)

            obj.j =10;

            obj.k=10;

            obj.m=10;

            obj.n=10;

       //     obj.s =10;  //Errror Static data can be accessed by class names only

            Class1.x = 10;

         //   obj.y = 10; //Errror Static data can be accessed by class names only

            Class1.y = 10;

        }

    }

Now lets try to copy the same code inside Main method and try to compile

[STAThread]

        static void Main()

        {

           //Access specifiers comes into picture only when you create object of class outside the class

            Class1 obj = new Class1();

       //     obj.i =10; //Error can’t access private data through object.

            obj.j =10;

      //      obj.k=10;     //Error can’t access protected data through object.

            obj.m=10;

            obj.n=10;

       //     obj.s =10;  //Errror Static data can be accessed by class names only

            Class1.x = 10;  //Error can’t access private data outside class

         //   obj.y = 10; //Errror Static data can be accessed by class names only

            Class1.y = 10;

        }

What if Main is inside another assembly

[STAThread]

        static void Main()

        {

           //Access specifiers comes into picture only when you create object of class outside the class

            Class1 obj = new Class1();

       //     obj.i =10; //Error can’t access private data through object.

            obj.j =10;

      //      obj.k=10;     //Error can’t access protected data through object.

     //     obj.m=10; // Error can’t access internal data outside assembly

    //      obj.n=10; // Error can’t access internal data outside assembly

       //     obj.s =10;  //Errror Static data can be accessed by class names only

            Class1.x = 10;  //Error can’t access private data outside class

         //   obj.y = 10; //Errror Static data can be accessed by class names only

            Class1.y = 10;

        }

In object-oriented software, complexity is managed by using **abstraction**.

**Abstraction** is a process that involves identifying the critical behavior of an object and eliminating irrelevant and complex details.

**Inheritance:**

Inheritance is a process of deriving the new class from already existing class

**C#** is a complete object oriented programming language. Inheritance is one of the primary concepts of object-oriented programming. It allows you to reuse existing code. Through effective use of inheritance, you can save lot of time in your programming and also reduce errors, which in turn will increase the quality of work and productivity. A simple example to understand inheritance in C#.

Using System;

Public class BaseClass

{

    Public BaseClass ()

    {

        Console.WriteLine ("Base Class Constructor executed");

    }

    Public void Write ()

    {

        Console.WriteLine ("Write method in Base Class executed");

    }

}

Public class ChildClass: BaseClass

{

    Public ChildClass ()

    {

        Console.WriteLine("Child Class Constructor executed");

    }

    Public static void Main ()

    {

        ChildClass CC = new ChildClass ();

        CC.Write ();

    }

}

In the Main () method in ChildClass we create an instance of childclass. Then we call the write () method. If you observe the ChildClass does not have a write() method in it. This write () method has been inherited from the parent BaseClass.

The output of the above program is   
  
**Output:**

  Base Class Constructor executed  
  Child Class Constructor executed  
  Write method in Base Class executed

this output proves that when we create an instance of a child class, the base class constructor will automatically be called before the child class constructor. So in general Base classes are automatically instantiated before derived classes.

In C# the syntax for specifying BaseClass and ChildClass relationship is shown below. The base class is specified by adding a colon, ":", after the derived class identifier and then specifying the base class name.

**Syntax:**  class **ChildClassName**: **BaseClass**  
              {  
                   //Body  
              }

C# supports single class inheritance only. What this means is, your class can inherit from only one base class at a time. In the code snippet below, class C is trying to inherit from Class A and B at the same time. This is not allowed in C#. This will lead to a compile time

error: **Class 'C' cannot have multiple base classes: 'A' and 'B'**.

public class A

{

}

public class B

{

}

public class C : A, B

{

}

In C# Multi-Level inheritance is possible. Code snippet below demonstrates mlti-level inheritance. Class B is derived from Class A. Class C is derived from Class B. So class C, will have access to all members present in both Class A and Class B. As a result of multi-level inheritance Class has access to A\_Method(),B\_Method() and C\_Method().   
  
***Note:*** Classes can inherit from multiple interfaces at the same time. **Interview Question:**How can you implement multiple inheritance in C#? **Ans :**Using Interfaces. We will talk about interfaces in our later article.

Using System;

Public class A

{

    Public void A\_Method ()

    {

        Console.WriteLine ("Class A Method Called");

    }

}

Public class B: A

{

    Public void B\_Method ()

    {

        Console.WriteLine ("Class A Method Called");

    }

}

Public class C: B

{

    Public void C\_Method ()

    {

        Console.WriteLine ("Class A Method Called");

    }

    Public static void Main ()

    {

        C C1 = new C ();

        C1.A\_Method ();

        C1.B\_Method ();

        C1.C\_Method ();

    }

}

When you derive a class from a base class, the derived class will inherit all members of the base class except constructors. In the code snippet below class B will inherit both M1 and M2 from Class A, but you cannot access M2 because of the private access modifier. Class members declared with a private access modifier can be accessed only with in the class. We will talk about access modifiers in our later article.   
  
**Common Interview Question:** Are private class members inherited to the derived class?

**Ans:** Yes, the private members are also inherited in the derived class but we will not be able to access them. Trying to access a private base class member in the derived class will report a compile time error.

Using System;

Public class A

{

Public void M1 ()

{

}

Private void M2 ()

{

}

}

Public class B: A

{

Public static void Main ()

{

B B1 = new B ();

B1.M1 ();

//Error, Cannot access private member M2

//B1.M2 ();

}

}

Method Hiding and Inheritance We will look at an example of how to **hide** a method in C#. The Parent class has a write () method which is available to the child class. In the child class I have created a new write () method. So, now if I create an instance of child class and call the write () method, the child class write () method will be called. The child class is hiding the base class write () method. This is called method hiding.   
  
If we want to call the parent class write () method, we would have to type cast the child object to Parent type and then call the write () method as shown in the code snippet below.

Using System;

Public class Parent

{

    Public void Write ()

    {

        Console.WriteLine ("Parent Class write method");

    }

}

Public class Child: Parent

{

    Public new void Write ()

    {

        Console.WriteLine ("Child Class write method");

    }

    Public static void Main ()

    {

        Child C1 = new Child ();

        C1.Write ();

        //Type caste C1 to be of type Parent and call Write () method

        ((Parent) C1).Write ();

    }

}

**Polymorphism:**

When a message can be processed in different ways is called polymorphism. Polymorphism means many forms.

Polymorphism is one of the fundamental concepts of OOP.

**Polymorphism provides following features:**

* It allows you to invoke methods of derived class through base class reference during runtime.
* It has the ability for classes to provide different implementations of methods that are called through the same name.

**Polymorphism is of two types:**

1. Compile time polymorphism/Overloading
2. Runtime polymorphism/Overriding

**Compile Time Polymorphism**

Compile time polymorphism is method and operators overloading. It is also called early binding.

In method overloading method performs the different task at the different input parameters.

**Runtime Time Polymorphism**

Runtime time polymorphism is done using inheritance and virtual functions. Method overriding is called runtime polymorphism. It is also called late binding.

When **overriding** a method, you change the behavior of the method for the derived class.  **Overloading**a method simply involves having another method with the same prototype.

**Caution:**Don't confused method overloading with method overriding, they are different, unrelated concepts. But they sound similar.

Method overloading has nothing to do with inheritance or virtual methods.

**Following are examples of methods having different overloads:**

void area(int side);

void area(int l, int b);

void area(float radius);

**Practical example of Method Overloading (Compile Time Polymorphism)**

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 ();

        }

    }

}

**Note:**In the code if you observe display method is called two times. Display method will work according to the number of parameters and type of parameters.

**When and why to use method overloading**

Use method overloading in situation where you want a class to be able to do something, but there is more than one possibility for what information is supplied to the method that carries out the task.

You should consider overloading a method when you for some reason need a couple of methods that take different parameters, but conceptually do the same thing.

**Method overloading showing many forms.**

using System;

namespace method\_overloading\_polymorphism

{

    Class Program

    {

        Public class Shape

        {

            Public void Area (float r)

            {

                float a = (float)3.14 \* r;

                // here we have used function overload with 1 parameter.

                Console.WriteLine ("Area of a circle: {0}",a);

            }

            Public void Area(float l, float b)

            {

                float x = (float)l\* b;

                // here we have used function overload with 2 parameters.

                Console.WriteLine ("Area of a rectangle: {0}",x);

            }

            public void Area(float a, float b, float c)

            {

                float s = (float)(a\*b\*c)/2;

                // here we have used function overload with 3 parameters.

                Console.WriteLine ("Area of a circle: {0}", s);

            }

        }

        Static void Main (string[] args)

        {

            Shape ob = new Shape ();

            ob.Area(2.0f);

            ob.Area(20.0f,30.0f);

            ob.Area(2.0f,3.0f,4.0f);

            Console.ReadLine ();

        }

    }

}

**Things to keep in mind while method overloading**

If you use overload for method, there are couple of restrictions that the compiler imposes.

The rule is that overloads must be different in their signature, which means the name and the number and type of parameters.

There is no limit to how many overload of a method you can have. You simply declare them in a class, just as if they were different methods that happened to have the same name.

**Method Overriding:**

Whereas **Overriding** means changing the functionality of a method without changing the signature. We can override a function in base class by creating a similar function in derived class. This is done by using virtual/override keywords.  
  
Base class method has to be marked with virtual keyword and we can override it in derived class using override keyword.  
  
Derived class method will completely overrides base class method i.e. when we refer base class object created by casting derived class object a method in derived class will be called.

Example:

// Base class  
public class BaseClass  
{  
public virtual void Method1()  
{  
Console.Write("Base Class Method");  
}  
}  
// Derived class  
public class DerivedClass : BaseClass  
{  
public override void Method1()  
{  
Console.Write("Derived Class Method");  
}  
}  
// Using base and derived class  
public class Sample  
{  
public void TestMethod()  
{  
// calling the overriden method  
DerivedClass objDC = new DerivedClass();   
objDC.Method1();  
 // calling the baesd class method  
BaseClass objBC = (BaseClass)objDC;   
objDC.Method1();  
}  
}

Output  
---------------------

Derived Class Method

Derived Class Method

**Difference between Abstraction and Encapsulation**  
  
Abstraction is a process. It is the act of identifying the relevant qualities and behaviors an object should possess. Encapsulation is the mechanism by which the abstraction is implemented.

|  |  |
| --- | --- |
| Abstraction | Encapsulation |
| Abstraction solves the problem in the design level. | Encapsulation solves the problem in the implementation level. |
| Abstraction is used for hiding the unwanted data and giving only relevant data. | Encapsulation is hiding the code and data into a single unit to protect the data from outer world. |
| 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. |
| 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 |

**Constructors and Destructors:**

Classes have complicated internal structures, including data and functions, object initialization and cleanup for classes is much more complicated than it is for simple data structures. Constructors and destructors are special member functions of classes that are used to construct and destroy class objects. Construction may involve memory allocation and initialization for objects. Destruction may involve cleanup and deallocation of memory for objects.

* Constructors and destructors do not have return types nor can they return values.
* References and pointers cannot be used on constructors and destructors because their addresses cannot be taken.
* Constructors cannot be declared with the keyword virtual.
* Constructors and destructors cannot be declared const, or volatile.
* Unions cannot contain class objects that have constructors or destructors.

Constructors and destructors obey the same access rules as member functions. For example, if you declare a constructor with protected access, only derived classes and friends can use it to create class objects.

The compiler automatically calls constructors when defining class objects and calls destructors when class objects go out of scope. A constructor does not allocate memory for the class object it’s this pointer refers to, but may allocate storage for more objects than its class object refers to. If memory allocation is required for objects, constructors can explicitly call the new operator. During cleanup, a destructor may release objects allocated by the corresponding constructor. To release objects, use the delete operator.

**Example of Constructor**

class C

{

       private int x;

       private int y;

       public C (int i, int j)

       {

                 x = i;

                 y = j;

       }

       public void display ()

       {

               Console.WriteLine(x + "i+" + y);

       }

}  
  
**Example of Destructor**

class D

{

        public D ()

        {

            // constructor

        }

        ~D ()

        {

           // Destructor

        }

}

What Is Constructor.

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 check below sample method for constructor

|  |
| --- |
| class SampleA  {  public SampleA()  {  Console.WriteLine("Sample A Test Method");  }  } |

**Types of Constructors**

Basically constructors are 5 types those are

      1.    Default Constructor

      2.    Parameterized Constructor

      3.    Copy Constructor

      4.    Static Constructor

      5.    Private Constructor

**Default Constructor**

A constructor without having any parameters called default constructor. In this constructor every instance of the class will be initialized without any parameter values like as shown below

|  |
| --- |
| 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 |

**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 |

**Constructor Overloading**

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 |

**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. Check below example for this

|  |
| --- |
| 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 |

**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.

**Private Constructor**

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","to Aspdotnet-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","to Aspdotnet-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

what is CLR in .net ?

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. All delegates are implicitly derived from the **System.Delegate**class.

1. Delegates are similar to C++ function pointers, but are type safe.
2. Delegates allow methods to be passed as parameters.
3. Delegates can be used to define callback methods.
4. Delegates can be chained together; for example, multiple methods can be called on a single event.
5. Methods don't need to match the delegate signature exactly.
6. Using a delegate allows the programmer to encapsulate a reference to a method inside a delegate object. The delegate object can then be passed to code that can call the referenced method, without having to know at compile time which method will be invoked.
7. An interesting and useful property of a delegate is that it does not know or care about the class of the object that it references. Any object will do; all that matters is that the method's argument types and return type match the delegate's. This makes delegates perfectly suited for "anonymous" invocation.

**Syntax of Delegate & Methods Declaration**

|  |
| --- |
| public delegate int Delegatmethod(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;  }  } |

Delegates are two types

      -   Single Cast Delegates

      -  Multi Cast Delegates

**Single Cast Delegates**

Single cast delegate means which hold address of single method like as explained in above example.

**Multicast Delegates**

Multi cast 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 list

**Complete 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();  MultiDelegate del = Sampleclass.Add;  del += Sampleclass.Sub;  del += Sampleclass.Mul;  del(10, 5);  Console.ReadLine();  }  } |