Unit -4
Advanced C#

Course Contents [14 Hrs.]

Delegates; Events; Lambda Expressions; Exception Handling; Introduction to LINQ; Working with Databases; Web Applications using ASP.NET

Delegates

A delegate is an object that knows how to call a method. A delegate type defines the kind of method that delegate instances can call. Specifically, it defines the method's return type and its parameter types.

Delegates are especially used for implementing events and the callback methods. All delegates are implicitly derived from the **System.Delegate** class. It provides a way, which tells which method is to be called when an event is triggered. For example, if you click an **Button** on a form **(Windows Form application),** the program would call a specific method.

In simple words, it is a type that represents references to methods with a particular parameter list and return type and then calls the method in a program for execution when it is needed.

Declaring Delegates

Delegate type can be declared using the **delegate** keyword. Once a delegate is declared, delegate instance will refer and call those methods whose return type and parameter-list matches with the delegate declaration.

Syntax:

[modifier] delegate [return_type] [delegate_name] ([parameter_list]);

modifier: It is the required modifier which defines the access of delegate and it is optional to use.

delegate: It is the keyword which is used to define the delegate.

return_type: It is the type of value returned by the methods which the delegate will be going to call. It can be void. A method must have the same return type as the delegate.

delegate_name: It is the user-defined name or identifier for the delegate.

parameter_list: This contains the parameters which are required by the method when called through the delegate.

Example

public delegate int DelegateTest(int x, int y, int z);

Note: A delegate will call only a method that agrees with its signature and return type. A method can be a static method associated with a class or can be an instance method associated with an object, it doesn't matter.

Instantiation & Invocation of Delegates

Once a delegate type is declared, a delegate object must be created with the **new** keyword and be associated with a particular method. When creating a delegate, the argument passed to the **new** expression is written similar to a method call, but without the arguments to the method.

Syntax

[delegate_name] [instance_name] = new [delegate_name](calling_method_name);

Example

```
DelegateTest obj = new DelegateTest (MyMethod);
// here,
// " DelegateTest" is delegate name.
// " obj" is instance_name
// " MyMethod" is the calling method.
```

The following defines a delegate type called **Transformer**:

delegate int Transformer (int x);

Transformer is **compatible** with any method with an **int return type and a single int parameter**, such as this:

```
static int Square (int x) { return x * x; }
or
static int Square (int x) => x * x;
```

Assigning a method to a delegate variable creates a delegate instance:

Transformer t = new Transformer (Square);

Or

Transformer t = Square;

which can be invoked in the same way as a method:

int answer = t(3); // answer is 9

Example of simple delegate is shown below:

```
public delegate int MyDelegate(int x);
class DelegateTest {
    static int MyMethod(int x) {
        return x * x;
    }

    static void Main(string[] agrs) {
        MyDelegate del = new MyDelegate(MyMethod);
        int res = del(5); //25
        Console.WriteLine("Result is : "+res);
        Console.ReadKey(); }
}
```

Output:

```
Result is : 25
```

Example 2

```
using System;
delegate int NumberChanger(int n);
class TestDelegate {
    static int num = 10;
    public static int AddNum(int p) {
        num += p;
        return num;
    }
    public static int MultNum(int q) {
        num *= q;
        return num;
    }
    public static int getNum() {
        return num;
    }
    static void Main(string[] args) {
        //create delegate instances
        NumberChanger nc1 = new NumberChanger(AddNum);
        NumberChanger nc2 = new NumberChanger(MultNum);

        //calling the methods using the delegate objects
        nc1(25);
        Console.WriteLine("Value of Num: {0}", getNum());
        nc2(5);
        Console.WriteLine("Value of Num: {0}", getNum());
        Console.ReadKey();
        }
    }
```

Output:

```
Value of Num: 35
Value of Num: 175
```

Example 3

```
class Test {
    public delegate void addnum(int a, int b);
    public delegate void subnum(int a, int b);

    // method "sum"
    public void sum(int a, int b) {
        Console.WriteLine("(100 + 40) = {0}", a + b);
}
// method "subtract"
public void subtract(int a, int b) {
    Console.WriteLine("(100 - 60) = {0}", a - b);
}

// Main Method
public static void Main(String[] args) {
    addnum del_obj1 = new addnum(obj.sum);
    subnum del_obj2 = new subnum(obj.subtract);

    // pass the values to the methods by delegate object
    del_obj1(100, 40);
    del_obj2(100, 60);

    // These can be written as using "Invoke" method
    // del_obj1.Invoke(100, 40);
    // del_obj2.Invoke(100, 60);
    }
}
```

Output:

```
(100 + 40) = 140
(100 - 60) = 40
```

Multicast Delegates

All delegate instances have multicast capability. **This means that a delegate instance can reference not just a single target method, but also a list of target methods.** The + and += operators combine delegate instances.

For example:

```
SomeDelegate d = SomeMethod1;
d += SomeMethod2;
```

The last line is functionally the same as:

```
d = d + SomeMethod2;
```

Invoking d will now call both SomeMethod1 and SomeMethod2. **Delegates are invoked in the order they are added.**

The - and -= operators remove the right delegate operand from the left delegate operand.

For example:

d -= SomeMethod1;

Invoking d will now cause only SomeMethod2 to be invoked.

Calling + or += on a delegate variable with a null value works, and it is equivalent to assigning the variable to a new value:

SomeDelegate d = null;

d += SomeMethod1; // Equivalent (when d is null) to d = SomeMethod1;

Similarly, calling -= on a delegate variable with a single target is equivalent to assigning null to that variable.

Example of multicasting delegates

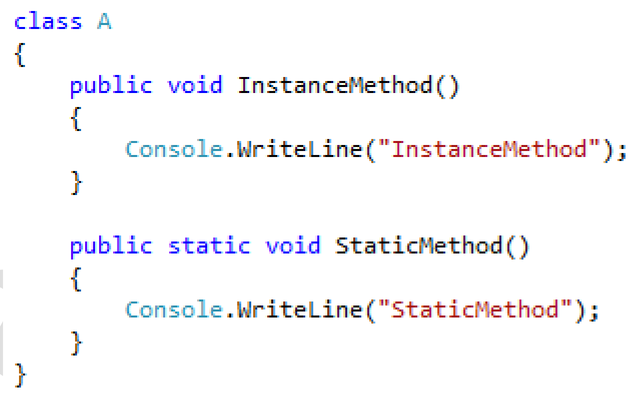
```
using System;
delegate int NumberChanger(int n);
    class TestDelegate {
        static int num = 10;

        public static int AddNum(int p) {
            num += p; return num;
        }

        public static int MultNum(int q) {
            num *= q; return num;
        }

        public static int getNum() {
                return num;
        }
        static void Main(string[] args) {
        //create delegate instances
        NumberChanger nc;
        NumberChanger nc1 = new NumberChanger(AddNum);
        NumberChanger nc2 = new NumberChanger(MultNum);

        nc = nc1;
        nc += nc2;

        //calling multicast nc(5);
        Console.WriteLine("Value of Num: {0}", getNum()); Console.ReadKey();
        }
}
```

Output

```
Value of Num: 75
```

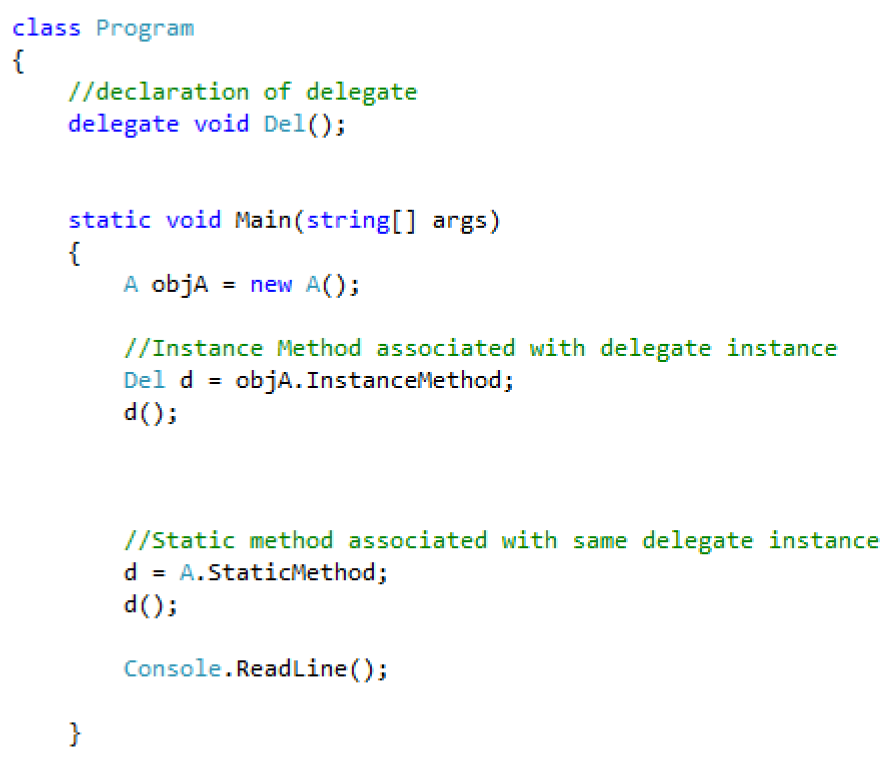
Delegates Mapping with Instance and Static Method

Created ClassA contains instance & static method,



```
class A
{
    public void InstanceMethod()
    {
        Console.WriteLine("InstanceMethod");
    }

    public static void StaticMethod()
    {
        Console.WriteLine("StaticMethod");
    }
}
```

Above class methods are used using delegate,



```
class Program
{
    //declaration of delegate
    delegate void Del();


    static void Main(string[] args)
    {
        A objA = new A();

        //Instance Method associated with delegate instance
        Del d = objA.InstanceMethod;
        d();



        //Static method associated with same delegate instance
        d = A.StaticMethod;
        d();

        Console.ReadLine();

    }
```

Output

```
Instance Method
Static Method
```

So using delegate, we can associate instance and static method under same delegate instance.

Delegates Vs Interfaces in C#

S.No.	**Delegate**	**Interface**
1	It could be a method only.	It contains both methods and properties.
2	It can be applied to one method at a time.	If a class implements an interface, then it will implement all the methods related to that interface.
3	If a delegate available in your scope you can use it.	Interface is used when your class implements that interface, otherwise not.
4	Delegates can be implemented any number of times.	Interface can be implemented only one time.

5	It is used to handling events.	It is not used for handling events.
6	When you access the method using delegates, you do not require any access to the object of the class where the method is defined.	When you access the method, you need the object of the class, which implemented an interface.
7	It does not support inheritance.	It supports inheritance.
8	It created at run time.	It created at compile time.
9	It can implement any method that provides the same signature with the given delegate.	If the method of interface implemented, then the same name and signature method override.
10	It can wrap any method whose signature is similar to the delegate and does not consider which from class it belongs.	A class can implement any number of interfaces, but can only override those methods, which belongs to the interfaces.

Delegate Compatibility

Type compatibility

Delegate types are all incompatible with one another, even if their signatures are the same:

```
delegate void D1();
delegate void D2();
...

D1 d1 = Method1;
D2 d2 = d1;                           // Compile-time error
```

The following, however, is permitted:

```
D2 d2 = new D2 (d1);
```

Delegate instances are considered equal if they have the same method targets:

```
delegate void D();
...

D d1 = Method1;
D d2 = Method1;
Console.WriteLine (d1 == d2);         // True
```

Multicast delegates are considered equal if they reference the same methods *in the same order.*

Parameter compatibility

When you call a method, you can supply arguments that have more specific types than the parameters of that method. This is ordinary polymorphic behaviour. For exactly the same reason, a delegate can have more specific parameter types than its method target. This is called **contra variance.**

Here's an example:

```
delegate void StringAction (string s);

class Test
{
  static void Main()
  {
    StringAction sa = new StringAction (ActOnObject);
    sa ("hello");
  }

  static void ActOnObject (object o) => Console.WriteLine (o);   // hello
}
```

In this case, the String Action is invoked with an argument of type string. When the argument is then relayed to the target method, the argument gets implicitly up cast to an object.

Return type compatibility

If you call a method, you may get back a type that is more specific than what you asked for. This is ordinary polymorphic behaviour. For exactly the same reason, a delegate's target method may return a more specific type than described by the delegate. This is called **covariance**. For example:

```
delegate object ObjectRetriever();

class Test
{
  static void Main()
  {
    ObjectRetriever o = new ObjectRetriever (RetrieveString);
    object result = o();
    Console.WriteLine (result);      // hello
  }
  static string RetrieveString() => "hello";
}
```

Generic Delegate Types

A delegate type may contain generic type parameters. For example:

public delegate T Transformer (T arg);

With this definition, we can write a generalized Transform utility method that works on any type:

```
public class Util
{
  public static void Transform<T> (T[] values, Transformer<T> t)
  {
    for (int i = 0; i < values.Length; i++)
      values[i] = t (values[i]);
  }
}

class Test
{
  static void Main()
  {
    int[] values = { 1, 2, 3 };
    Util.Transform (values, Square);      // Hook in Square
    foreach (int i in values)
      Console.Write (i + "  ");           // 1   4   9
  }

  static int Square (int x) => x * x;
}
```

Func and Action Delegates

The Func and Action generic delegates were introduced in the .NET Framework version 3.5. Whenever we want to use delegates in our examples or applications, typically we use the following procedure:

- Define a custom delegate that matches the format of the method.
- Create an instance of a delegate and point it to a method.
- Invoke the method.

But, using these 2 Generics delegates we can simply eliminate the above procedure. Since both the delegates are generic, you will need to specify the underlying types of each parameter as well while pointing it to a function. For example

Action<type,type,type……>

Action<>

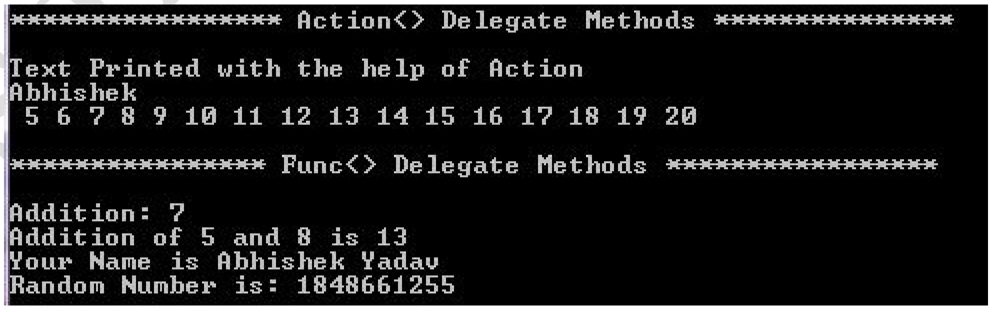
- This Action<> generic delegate; points to a method that takes up to 16 Parameters and returns void.

Func<>

- The generic Func<> delegate is used when we want to point to a method that returns a value.
- This delegate can point to a method that takes up to 16 Parameters and returns a value.
- Always remember that the final parameter of Func<> is always the return value of the method. (For example, Func< int, int, string>, this version of the Func<> delegate will take 2 int parameters and returns a string value.)

Example is shown below

```
    class MethodCollections {
        //Methods that takes parameters but returns nothing:
        public static void PrintText() {
            Console.WriteLine("Text Printed with the help of Action");
        }
        public static void PrintNumbers(int start, int target) {
                for (int i = start; i <= target; i++) {
                        Console.Write(" {0}",i);
                }
                Console.WriteLine();
        }
        public static void Print(string message) {
                Console.WriteLine(message);
        }
        //Methods that takes parameters and returns a value:
        public static int Addition(int a, int b) {
                return a + b;
        }
        public static string DisplayAddition(int a, int b) {
                return string.Format("Addition of {0} and {1} is {2}",a,b,a+b);
        }
        public static string ShowCompleteName(string firstName, string las tName) {
                return string.Format("Your Name is {0} {1}",firstName,lastName);
        }
        public static int ShowNumber() {
                Random r = new Random();
                return r.Next();
        }
}
class Program {
        static void Main(string[] args) {
                Action printText = new Action(MethodCollections.PrintText);
                Action<string> print = new Action<string>(MethodCollections.Print);
Action<int, int> printNumber = new Action<int, int>(MethodColl ections.PrintNumbers);
                Func<int, int,int> add1 = new Func<int, int, int>(MethodCollec tions.Addition);
Func<int, int, string> add2 = new Func<int, int, string>(Metho dCollections.DisplayAddition);
Func<string, string, string> completeName = new Func<string, string, string>(MethodCollections.SHowCompleteName);
Func<int> random = new Func<int>(MethodCollections.ShowNumber);
                Console.WriteLine("\n*****************Action<> Delegate Method *******\n");
                printText(); //Parameter: 0 , Returns: nothing
                print("Abhishek"); //Parameter: 1 , Returns: nothing
                printNumber(5, 20); //Parameter: 2 , Returns: nothing
                Console.WriteLine();
                Console.WriteLine("********** Func<> Delegate Methods ** *************\n");

                int addition = add1(2, 5); //Parameter: 2 , Returns: int
                string addition2 = add2(5, 8); //Parameter:2 ,Returns: string
        string name = completeName("Abhishek", "Yadav"); //Parameter:2 , Returns: string
        int randomNumbers = random(); ////Parameter: 0 , Returns: int
        Console.WriteLine("Addition: {0}",addition);
        Console.WriteLine(addition2);
        Console.WriteLine(name);
        Console.WriteLine("Random Number is: {0}",randomNumbers);
        Console.ReadLine();
        }
}
```



```
***************** Action<> Delegate Methods ****************

Text Printed with the help of Action
Abhishek
 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

***************** Func<> Delegate Methods *****************

Addition: 7
Addition of 5 and 8 is 13
Your Name is Abhishek Yadav
Random Number is: 1848661255
```