Delegate in C#

A delegate is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance.

Delegates are used to pass methods as arguments to other methods. Event handlers are nothing more than methods that are invoked through delegates. You create a custom method, and a class such as a windows control can call your method when a certain event occurs. The following example shows syntax for delegate declaration:

```
<Access Modifier> delegate <return type> <delegate name> (arguments list);
```

For example:

```
public delegate int PerformCalculation(int x, int y);
```

Properties of Delegates

- Delegates are similar to C++ function pointers, but delegates are fully object-oriented, and unlike C++ pointers to member functions, delegates encapsulate both an object instance and a method.
- Delegates allow methods to be passed as parameters.
- Delegates can be used to define callback methods.
- Delegates can be chained together; for example, multiple methods can be called on a single event.
- Methods do not have to match the delegate type exactly.
- C# version 2.0 introduced the concept of anonymous methods, which allow code blocks to be passed as parameters in place of a separately defined method. C# 3.0 introduced lambda expressions as a more concise way of writing inline code blocks. Both anonymous methods and lambda expressions (in certain contexts) are compiled to delegate types. Together, these features are now known as anonymous functions.

Rules of using Delegates in C#

- A delegate in C# is a user-defined type and hence before invoking a method using delegate, we must have to define that delegate first.
- The signature of the delegate must match the signature of the method, the delegate points to otherwise we will get a compiler error. This is the reason why delegates are called as type-safe function pointers.
- A Delegate is similar to a class. Means we can create an instance of a delegate and when we do so, we need to pass the method name as a parameter to the delegate constructor, and it is the function the delegate will point to
- Delegates syntax look very much similar to a method with a delegate keyword.

Types of Delegates

The Delegates in C# are classified into two types:

1. Single cast delegate: A delegate is used for invoking a single method then it is called a single cast delegate or unicast delegate.
2. Multicast delegate: A delegate is used for invoking multiple methods then it is known as the multicast delegate.

Using Delegates

The following example declares a `delegate` named `Del` that can encapsulate a method that takes a `string` as an argument and returns `void`:

```
public delegate void Del(string message);
```

A delegate object is normally constructed by providing the name of the method the delegate will wrap, or with an anonymous function. Once a delegate is instantiated, a method call made to the delegate will be passed by the delegate to that method. The parameters passed to the delegate by the caller are passed to the method, and the return value, if any, from the method is returned to the caller by the delegate. This is known as invoking the delegate. An instantiated delegate can be invoked as if it were the wrapped method itself. For example:

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}

. . .

// Instantiate the delegate.
Del handler = DelegateMethod;
// Call the delegate.
handler("Hello World");

. . .
```

Delegate types are derived from the Delegate class in .NET. Because the instantiated delegate is an object, it can be passed as a parameter, or assigned to a property. This allows a method to accept a delegate as a parameter, and call the delegate at some later time. This is known as an asynchronous **callback**, and is a common method of notifying a caller when a long process has completed. When a delegate is used in this fashion, the code using the delegate does not need any knowledge of the implementation of the method being used. The functionality is similar to the encapsulation interfaces provide.

Another common use of callbacks is defining a custom comparison method and passing that delegate to a sort method. It allows the caller's code to become part of the sort algorithm. The following example method uses the `Del` type as a parameter:

```
public static void MethodWithCallback(int param1, int param2, Del callback)
{
    callback("The number is: " + (param1 + param2).ToString());
}
```

You can then pass the delegate created above to that method:

```
MethodWithCallback(1, 2, handler);
```

and receive the following output to the console:

```
The number is: 3
```

Using the delegate as an abstraction, `MethodWithCallback` does not need to call the console directly—it does not have to be designed with a console in mind. What `MethodWithCallback` does is simply prepare a string and pass the string to another method. This is especially powerful since a delegated method can use any number of parameters.

When a delegate is constructed to wrap an instance method, the delegate references both the instance and the method. A delegate has no knowledge of the instance type aside from the method it wraps, so a delegate can refer to any type of object as long as there is a method on that object that matches the delegate signature. When a delegate is constructed to wrap a static method, it only references the method. Consider the following declarations:

```
public class MethodClass
{
    public void Method1(string message) { }
    public void Method2(string message) { }
}
```

Along with the static `DelegateMethod` shown previously, we now have three methods that can be wrapped by a `Del` instance.

A delegate can call more than one method when invoked. This is referred to as **multicasting**. To add an extra method to the delegate's list of methods—the invocation list—simply requires adding two delegates using the addition or addition assignment operators ('+' or '+='). For example:

```
var obj = new MethodClass();
Del d1 = obj.Method1;
Del d2 = obj.Method2;
Del d3 = DelegateMethod;
//Both types of assignment are valid.
Del allMethodsDelegate = d1 + d2;
allMethodsDelegate += d3;
```

At this point `allMethodsDelegate` contains three methods in its invocation list—`Method1`, `Method2`, and `DelegateMethod`. The original three delegates, `d1`, `d2`, and `d3`, remain unchanged. When `allMethodsDelegate` is invoked, all three methods are called in order. If the delegate uses reference parameters, the reference is passed sequentially to each of the three methods in turn, and any changes by one method are visible to the next method. When any of the methods throws an exception that is not caught within the method, that exception is passed to the caller of the delegate and no subsequent methods in the invocation list are called. **If the delegate has a return value and/or out parameters, it returns the return value and parameters of the last method invoked**. To remove a method from the invocation list, use the subtraction or subtraction assignment operators (- or -=). For example:

```
//remove Method1
allMethodsDelegate -= d1;
// copy AllMethodsDelegate while removing d2
```

```
Del oneMethodDelegate = allMethodsDelegate - d2;
```

Because delegate types are derived from `System.Delegate`, the methods and properties defined by that class can be called on the delegate. For example, to find the number of methods in a delegate's invocation list, you may write:

```
int invocationCount = d1.GetInvocationList().GetLength(0);
```

Delegates with more than one method in their invocation list derive from `MulticastDelegate`, which is a subclass of `System.Delegate`. The above code works in either case because both classes support `GetInvocationList`.

Example:

```
namespace MulticastDelegateDemo
{
    public delegate int MathDelegate(int No1, int No2);
    public class Program
    {
        public static int M {get; set;}
        public static int Add(int x, int y)
        {
            M = 0;
            M += (x + y);
            Console.WriteLine("THE SUM IS : " + (x + y));
            return M;
        }
        public static int Sub(int x, int y)
        {
            M -= (x - y);
            Console.WriteLine("THE SUB IS : " + (x - y));
            return M;
        }
        public int Mul(int x, int y)
        {
            M *= (x * y);
            Console.WriteLine("THE MUL IS : " + (x * y));
            return M;
        }
        public int Div(int x, int y)
        {
            M += (x / y);
            Console.WriteLine("THE DIV IS : " + (x / y));
            return M;
        }
        static void Main(string[] args)
        {
            Program p = new Program();
            MathDelegate del1 = new MathDelegate(Add);
            MathDelegate del2 = new MathDelegate(Program.Sub);
            MathDelegate del3 = p.Mul;
```

```
            MathDelegate del4 = new MathDelegate(p.Div);
        //In this example del5 is a multicast delegate. We can use +(plus)
        // operator to chain delegates together and -(minus) operator to remove.
            MathDelegate del5 = del1 + del2 + del3 + del4;
            Console.WriteLine("GRAND TOTAL : " + del5.Invoke(5, 6));
            Console.WriteLine();
            del5 -= del2;
            Console.WriteLine("GRAND TOTAL : " + del5(5, 6));
            Console.ReadKey();
        }
    }
}
//Output:
//     THE SUM IS : 11
//     THE SUB IS : -1
//     THE MUL IS : 30
//     THE DIV IS : 0
//     GRAND TOTAL : 40
//
//     THE SUM IS : 11
//     THE MUL IS : 30
//     THE DIV IS : 0
//     GRAND TOTAL : 41
```