By: Mukunda Paudel
paudelmuku@gmail.com

Delegates and Events

A function can have one or more parameters of different data types, but what if you want to pass a function itself as a parameter? How does C# handle the callback functions or event handler?

The answer is - **delegate**.

- All the delegates are implicitly derived from ***System.Delegate*** class.
- Is a type safe function pointer.
- It holds a reference of a method and then calls the method for execution.
- Signature of the delegates must match the signature of the function that delegates points to, otherwise you will get a compiler error. This is the reason delegates are called as type safe function pointer.
- Delegates is similar to class. You can create an instances of it, and when you do so, you pass in the function name as a parameter to the delegate constructor, and it is to this function the delegate will point to.
- Always define the delegates under namespace this is the best practice but you can define it under the class also.
- A delegate can be declared using **delegate** keyword followed by a function signature as shown below:

Syntax:

<access modifier> delegate <return type> <delegate_name>(<parameters>)

The following example declares a Calculate delegate.

Example: Declare delegate

```
public delegate void Calculate(int value);
```

Example:

```
namespace EverestLab
{
    //1.Defining the Delegate

//declare delegate (delegates is type safe pointer)
    public delegate void MyMessageFunctionDelegate(String msz);
    class Program
    {
        static void Main(string[] args)
        {
            // 2. Instantiating the Delegate / invoking delegates

           //MyMessageFunctionDelegates points to MyMessage
       MyMessageFunctionDelegate mszdel = new MyMessageFunctionDelegate(MyMessage);

       // 3.Call the Deligate (by passing require parameter values, so that internally
       the method which is bondedwith the delegates gets executed.

            mszdel("Hello i am Delegates...");

            // or
            // MyMessageFunctionDelegate mszdel = MyMessage; //another way of invoking
           // mszdel("Hello I am Delegates...");
          // mszdel.invoke("Hello I am Delegates…")
        }
        public static void MyMessage(string strmsz)
        {
            Console.WriteLine(strmsz + "\n Press Any key to Exit...");
            Console.ReadLine();
        }
    }
}
```

Multicasting of delegate

- The delegate can points to multiple methods. A delegate that points multiple methods is called a multicast delegate.
- Multicasting of delegate is an extension of the normal delegate (sometimes termed as Single Cast Delegate). It helps the user to point more than one method in a single call.

Properties:

- ✓ Delegates are combined and when you call a delegate then a complete list of methods is called.
- ✓ Delegates maintain three important pieces of information, as in the following:

1. The parameters of the method.

2. The address of the method it calls.

3. The return type of the method.

- ✓ The '+' or '+=' operator is used to add a function/methods to the delegate object and the '-'or '- =' operator is used to remove an existing function/methods from the delegate list / object.

Note: A multicasting of delegate should have a return type of Void otherwise it will throw a runtime exception.

C# program to perform addition and Subtraction of two Number using Delegates (Multicast)

```
namespace EverestLab
{
    //Intializing a Delegate Here(MULTICAST- Encapsulating more than one method)
    public delegate int Calculate(int value1, int value2);
    class Program
    {
        static void Main(string[] args)
        {
            ForAddition _foraddition = new ForAddition();
            //This is a MULTICAST Delegate
            Calculate add = new Calculate(_foraddition.add);
            Calculate sub = new Calculate(_foraddition.sub);
            Console.WriteLine("Addition of Two Numbers ::" + add(10, 20));
            Console.WriteLine("Subtraction of Two numbers :: " + sub(50, 25));
            Console.ReadLine();
        }
    }
       //Creating a Class with methods inorder to invoke the Delegate Object With Same
       //Signature(ReturnType+Prameters)
    class ForAddition
    {
        public int add(int value1, int value2)
        {
            return value1 + value2;
        }
        public int sub(int value1, int value2)
        {
            return value1 - value2;
        }
    }
}
```

As you can see in the above example, Print delegates becomes a multicast delegate because it points to two methods - add & sub. So invoking **_foraddition** will invoke all the methods sequentially.

- ✓ Delegate is also used with ***Event, Anonymous method, Func delegate, Action delegate.***

Events in C#

- ❖ In general terms, an event is something special that is going to happen.
- ❖ An event has a **publisher**, **subscriber**, notification and a **handler**.
- ❖ The class that sends (or *raises*) the event is called the ***publisher*** and the classes that receive (or *handle*) the event are called ***subscribers***
- ❖ Generally, UI controls use events extensively.
- ❖ An event is nothing but an encapsulated delegate.
- ❖ Defining an event is a two-step process.
  - ✓ First, you need to define a delegate type that will hold the list of methods to be called when the event is fired.
  - ✓ Next, you declare an event using the event keyword.
- ❖ Events enable a class or object to notify other classes or objects when something of interest occurs.
- ❖ The IDE provides an easy way to automatically add an empty event handler method and the code to subscribe to the event.

Events have the following properties:

- ❖ The publisher determines when an event is raised; the subscribers determine what action is taken in response to the event.
- ❖ An event can have multiple subscribers. A subscriber can handle multiple events from multiple publishers.
- ❖ Events that have no subscribers are never raised.
- ❖ Events are typically used to signal user actions such as button clicks or menu selections in graphical user interfaces.
- ❖ When an event has multiple subscribers, the event handlers are invoked synchronously when an event is raised. To invoke events asynchronously
- ❖ In the .NET Framework class library, events are based on the **EventHandler** delegate and the **EventArgs** base class.
- ❖ To declare an event, use the **event** keyword before declaring a variable of delegate type.

E.g. Delegate

```
public delegate void someEvent();

public someEvent someEvent;
```

E.g. Event Declaration

```
public delegate void someEvent();

public event someEvent someEvent;
```

Thus, a delegate becomes an event using the **event** keyword.

Can we use Events without Delegate?

No, Events use Delegates internally. Events are encapsulation over Delegates. There is already defined Delegate "EventHandler" that can be used like:

```
public event EventHandler MyEvents;
```

So, it also used Delegate Internally.

***And**, Event Handlers can't return a value. They are always void.*

Example:

Define an event to add that is associated to a single delegate DelEventHandler.

```
namespace EverestLab
{
    public delegate void DelEventHandler();
    class Program
    {
        public static event DelEventHandler add;
        static void Main(string[] args)
        {
            add += new DelEventHandler(SURKHET);
            add += new DelEventHandler(KATHMANDU);
            add += new DelEventHandler(POKHARA);
            add.Invoke();

            Console.ReadLine();
        }
        static void SURKHET()
        {
            Console.WriteLine("SURKHET");
        }

        static void KATHMANDU()
        {
            Console.WriteLine("KATHMANDU");
        }

        static void POKHARA()
        {
            Console.WriteLine("POKHARA");
        }
    }

}
```

O/P

SURKHET

KATHMANDU

POKHARA

In the main method, we associate the event with its corresponding event handler with a function reference. Here, we are also filling the delegate invocation lists with a couple of defined methods using the +=operator as well. Finally, we invoke the event via the Invoke method.

Index Attribute

- Index attribute is used by Entity Framework Migrations to create indexes on mapped database columns.
- Multi-column indexes are created by using the same index name in multiple attributes. The information in these attributes is then merged together to specify the actual database index.
- Entity Framework 6 provides the **[Index]** attribute to create an index on a particular column in the database, as shown below:

```
Public class Student
{
        [Key]
   public int ID { get; set; }
   public string Name { get; set; }

   [Index]
   public int RollNumber { get; set; }
}
```

Versioning in C#

- Versioning tells about the compatibility of new release of dot net libraries.
- As a developer who has created .NET libraries for public use, you've most likely been in situations where you have to roll out new updates.
- How you go about this process matters a lot as you need to ensure that there's a seamless transition of existing code to the new version of your library.
- there are several things to consider when creating a new release

- **Semantic versioning** (**SemVer** for short) is a naming convention applied to versions of your library to signify specific milestone events.
- Ideally, the version information you give your library should help developers determine the compatibility with their projects that make use of older versions of that same library.

The most basic approach to **SemVer** is the 3 component format

- ✓ **MAJOR version**
- ✓ **MINOR version and**
- ✓ **PATCH,**

Where,

MAJOR →is incremented when you make incompatible API changes
MINOR→ is incremented when you add functionality in a backwards-compatible manner
PATCH→ is incremented when you make backwards-compatible bug fixes

- ❖ There are also ways to specify other scenarios like pre-release versions etc. when applying version information to your .NET library.
- ❖ As you release new versions of your library, **backwards compatibility** with previous versions will most likely be one of your major concerns.
- ❖ A new version of your library is source compatible with a previous version if code that depends on the previous version can, when recompiled, work with the new version.

Application Configuration File

- ❖ As a .NET developer there's a very high chance you've encountered the **app.config** file present in most project types.
- ❖ This simple configuration file can go a long way into improving the rollout of new updates.
- ❖ You should generally design your libraries in such a way that information that is likely to change regularly is stored in the **app.config** file, this way when such information is updated the **config** file of older versions just needs to be replaced with the new one without the need for recompilation of the library.