Events

Events are user actions such as key press, clicks, mouse movements, etc., or some occurrence such as system generated notifications. Applications need to respond to events when they occur. For example, interrupts. Events are used for inter-process communication.

The class that sends or raises an event is called a Publisher and class that receives or handle the event is called "Subscriber".

Following are the key points about Event,

1. Event Handlers in C# return void and take two parameters.
2. The First parameter of Event - Source of Event means publishing object.
3. The Second parameter of Event - Object derived from EventArgs.
4. The publishers determines when an event is raised and the subscriber determines what action is taken in response.
5. An Event can have so many subscribers.
6. Events are basically used for the single user action like button click.
7. If an Event has multiple subscribers then event handlers are invoked synchronously.

Declaring Events

To declare an event inside a class, first of all, you must declare a delegate type for the even as:

public delegate string MyDelegate(string str);

then, declare the event using the event keyword –

event MyDelegate delg;

The preceding code defines a delegate named MyDelegate and an event named delg, which invokes the delegate when it is raised.

To declare an event inside a class, first a Delegate type for the Event must be declared like below:

public delegate void MyEventHandler(object sender, EventArgs e);

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.

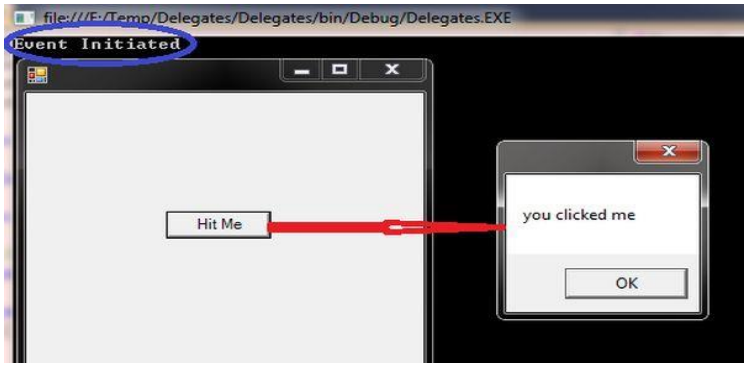
To illustrate the event, we are creating a console application. In this iteration, we will define an event to add that is associated to a single delegate **DelEventHandler**.

```
using System;
public delegate void DelEventHandler();
class Program {
        public static event DelEventHandler add;

        static void USA() {
                Console.WriteLine("USA");
        }
        static void India() {
                Console.WriteLine("India");
        }
        static void England() {
                Console.WriteLine("England");
        }
        static void Main(string[] args) {
                add += new DelEventHandler(USA);
                add += new DelEventHandler(India);
                add += new DelEventHandler(England);
                add.Invoke(); Console.ReadLine();
                }
        }
```

Implementing event in a button click

```
using System;
using System.Drawing;
using System.Windows.Forms;

//custom delegate public delegate void DelEventHandler();

class Program :Form {
        //custom event
        public event DelEventHandler add;

        public Program() {
```

```
        // design a button over form
        Button btn = new Button();
        btn.Parent = this;
        btn.Text = "Hit Me";
        btn.Location = new Point(100,100);

        //Event handler is assigned
        // the button click event
        btn.Click += new EventHandler(onClcik);
        add += new DelEventHandler(Initiate);

        //invoke the event
        add();
}
//call when event is fired
public void Initiate() {
        Console.WriteLine("Event Initiated");
}

//call when button clicked
public void onClick(object sender, EventArgs e) {
        MessageBox.Show("You clicked me");
}
static void Main(string[] args) {
        Application.Run(new Program());
        Console.ReadLine();
} }
```



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 below:

```
public event EventHandler MyEvents;
```

So, it also used Delegate Internally.

Anonymous Method in C#

An anonymous method is a method which doesn't contain any name which is introduced in C# 2.0. It is useful when the user wants to create an inline method and also wants to pass parameter in the anonymous method like other methods.

An Anonymous method is defined using the delegate keyword and the user can assign this method to a variable of the delegate type.

```
delegate(parameter_list){
    // Code..
};
```

Example:

```
using System;
class GFG {
    public delegate void petanim(string pet);

    // Main method
    static public void Main() {
        // An anonymous method with one parameter
        petanim p = delegate(string mypet) {
            Console.WriteLine("My favorite pet is: {0}", mypet);
        };
        p("Dog");
    }
}
```

Output:

```
My favorite pet is: Dog
```

You can also use an anonymous method as an event handler.

```
MyButton.Click += delegate(Object obj, EventArgs ev) {
        System.Windows.Forms.MessageBox.Show("Complete without error...!!");
}
```

Lambda Expressions

Lambda expressions in C# are used like anonymous functions, with the difference that in Lambda expressions you don't need to specify the type of the value that you input thus making it more flexible to use.

The '=>' is the lambda operator which is used in all lambda expressions. The Lambda expression is divided into two parts, the left side is the input and the right is the expression.

The Lambda Expressions can be of two types:

1. **Expression Lambda**: Consists of the input and the expression.
   Syntax: *input => expression;*
2. **Statement Lambda**: Consists of the input and a set of statements to be executed. It can be used along with delegates.
   Syntax: *input => { statements };*

Basic example of lambda expression:

```
class LambdaTest {
        static int test1() => 5;
        static int test2(int x) => x + 10;

        static void Main(string[] args) {
                int x=test1();
                int res = test2(x);
                Console.WriteLine("Result is: "+res);
        }
}
```

Output:

Result is: 15

Unlike an expression lambda, a statement lambda can contain multiple statements separated by semicolons. It is used with delegates.

```
delegate void ModifyInt(int input);
ModifyInt addOneAndTellMe = x => {
        int result = x + 1;
        Console.WriteLine(result);
};
```

Exception Handling

A try statement specifies a code block subject to error-handling or clean-up code. The try block must be followed by a catch block, a finally block, or both. The catch block executes when an error occurs in the try block. The finally block executes after execution leaves the try block (or if present, the catch block), to per- form clean-up code, whether or not an error occurred.

A catch block has access to an Exception object that contains information about the error. You use a catch block to either compensate for the error or re throw the exception. You re throw an exception if you merely want to log the problem, or if you want to re throw a new, higher-level exception type.

A finally block adds determinism to your program: the CLR endeavors to always execute it. It's useful for clean-up tasks such as closing network connections.

A try statement looks like this:

```
try
{
  ... // exception may get thrown within execution of this block
}
catch (ExceptionA ex)
{
  ... // handle exception of type ExceptionA
}
catch (ExceptionB ex)
{
  ... // handle exception of type ExceptionB
}
finally
{
  ... // cleanup code
}
```

Consider the following program:

```
class Test
{
  static int Calc (int x) => 10 / x;

  static void Main()
  {
    int y = Calc (0);
    Console.WriteLine (y);
  }
}
```

Because x is zero, the runtime throws a DivideByZeroException, and our program terminates. We can prevent this by catching the exception as follows:

```
class Test
{
  static int Calc (int x) => 10 / x;

  static void Main()
  {
    try
    {
      int y = Calc (0);

      Console.WriteLine (y);
    }
    catch (DivideByZeroException ex)
    {
      Console.WriteLine ("x cannot be zero");
    }
    Console.WriteLine ("program completed");
  }
}

OUTPUT:
x cannot be zero
program completed
```

The catch Clause

A catch clause specifies what type of exception to catch. This must either be System.Exception or a subclass of System.Exception.

You can handle multiple exception types with multiple catch clauses:

```
class Test
{
  static void Main (string[] args)
  {
    try
    {
      byte b = byte.Parse (args[0]);
      Console.WriteLine (b);
    }
    catch (IndexOutOfRangeException ex)
    {
      Console.WriteLine ("Please provide at least one argument");
    }
    catch (FormatException ex)
    {
      Console.WriteLine ("That's not a number!");
    }
    catch (OverflowException ex)
    {
      Console.WriteLine ("You've given me more than a byte!");
    }
  }
}
```

The finally Block

A finally block always executes—whether or not an exception is thrown and whether or not the try block runs to completion. finally blocks are typically used for clean-up code. A finally block executes either:

- After a catch block finishes.
- After control leaves the try block because of a jump statement (e.g., return or goto)
- After the try block ends

Throwing Exceptions

Exceptions can be thrown either by the runtime or in user code. In this example, Display throws a System.ArgumentNullException:

```
class Test
{
  static void Display (string name)
  {
    if (name == null)
      throw new ArgumentNullException (nameof (name));

    Console.WriteLine (name);
  }

  static void Main()
  {

     try { Display (null); }
     catch (ArgumentNullException ex)
     {
       Console.WriteLine ("Caught the exception");
     }
   }
}
```

Re-throwing an exception

You can capture and re-throw an exception as follows:

```
try {  ...  }
catch (Exception ex)
{
  // Log error
  ...
  throw;          // Rethrow same exception
}
```

Common Exception Types

System.ArgumentException

Thrown when a function is called with a bogus argument. This generally indi- cates a program bug.

System.ArgumentNullException

Subclass of ArgumentException that's thrown when a function argument is (unexpectedly) null.

System.ArgumentOutOfRangeException

Subclass of ArgumentException that's thrown when a (usually numeric) argu- ment is too big or too small. For example, this is thrown when passing a nega- tive number into a function that accepts only positive values.

System.InvalidOperationException

Thrown when the state of an object is unsuitable for a method to successfully execute, regardless of any particular argument values. Examples include read- ing an unopened file or getting the next element from an enumerator where the underlying list has been modified partway through the iteration.

System.NotSupportedException

Thrown to indicate that a particular functionality is not supported. A good example is calling the Add method on a collection for which IsReadOnly returns true.

System.NotImplementedException

Thrown to indicate that a function has not yet been implemented.

System.ObjectDisposedException

Thrown when the object upon which the function is called has been disposed.