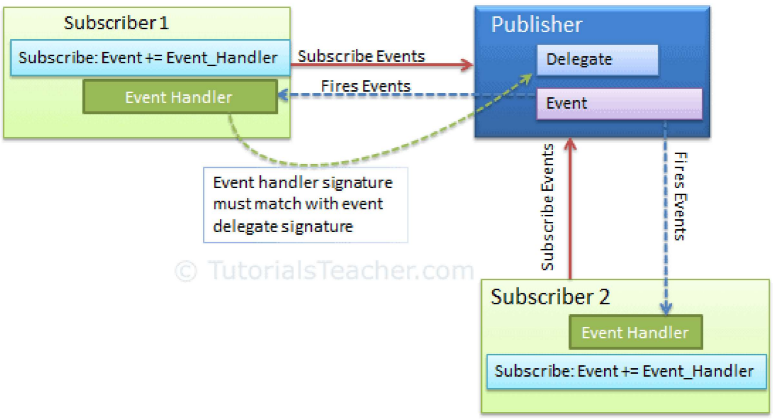
# C# - Events

An event is a notification sent by an object to signal the occurrence of an action. Events in .NET follow the observer design pattern.

The class who raises events is called Publisher, and the class who receives the notification is called Subscriber. There can be multiple subscribers of a single event. Typically, a publisher raises an event when some action occurred. The subscribers, who are interested in getting a notification when an action occurred, should register with an event and handle it.

In C#, an event is an encapsulated delegate. It is dependent on the delegate. The delegate defines the signature for the event handler method of the subscriber class.

The following figure illustrates the event in C#.




Event Publisher & Subscriber

## Declare an Event

An event can be declared in two steps:

1. Declare a delegate.
2. Declare a variable of the delegate with `event` keyword.

The following example shows how to declare an event in publisher class.

### Example: Declaring an Event

```
public delegate void Notify();  // delegate

public class ProcessBusinessLogic
{
    public event Notify ProcessCompleted; // event

}
```

In the above example, we declared a delegate `Notify` and then declared an event `ProcessCompleted` of delegate type `Notify` using "event" keyword in the `ProcessBusinessLogic` class. Thus, the `ProcessBusinessLogic` class is called the publisher. The `Notify` delegate specifies the signature for the `ProcessCompleted` event handler. It specifies that the event handler method in subscriber class must have a void return type and no parameters.

Now, let's see how to raise the `ProcessCompleted` event. Consider the following implementation.

Example: Raising an Event

```
public delegate void Notify();  // delegate

public class ProcessBusinessLogic
{
    public event Notify ProcessCompleted; // event

    public void StartProcess()
    {
        Console.WriteLine("Process Started!");
        // some code here..
        OnProcessCompleted();
    }

    protected virtual void OnProcessCompleted() //protected virtual method
    {
        //if ProcessCompleted is not null then call delegate
        ProcessCompleted?.Invoke();
    }
}
```

Above, the `StartProcess()` method calls the method `onProcessCompleted()` at the end, which raises an event. Typically, to raise an event, protected and virtual method should be defined with the name `On<EventName>`. Protected and virtual enable derived classes to override the logic for raising the event. However, A derived class should always call the `On<EventName>` method of the base class to ensure that registered delegates receive the event.

The `OnProcessCompleted()` method invokes the delegate using `ProcessCompleted?.Invoke();`. This will call all the event handler methods registered with the `ProcessCompleted` event.

The subscriber class must register to `ProcessCompleted` event and handle it with the method whose signature matches `Notify` delegate, as shown below.

Example: Consume an Event

```
class Program
{
    public static void Main()
    {
        ProcessBusinessLogic bl = new ProcessBusinessLogic();
        bl.ProcessCompleted += bl_ProcessCompleted; // register with an event
        bl.StartProcess();
    }

    // event handler
    public static void bl_ProcessCompleted()
    {
        Console.WriteLine("Process Completed!");
    }
}
```

Try it

Above, the `Program` class is a subscriber of the `ProcessCompleted` event. It registers with the event using += operator. Remember, this is the same way we add methods in the invocation list of multicast delegate. The `bl_ProcessCompleted()` method handles the event because it matches the signature of the `Notify` delegate.

## Built-in EventHandler Delegate

.NET Framework includes built-in delegate types EventHandler and EventHandler<TEventArgs> for the most common events. Typically, any event should include two parameters: the source of the event and event data. Use the `EventHandler` delegate for all events that do not include event data. Use EventHandler<TEventArgs> delegate for events that include data to be sent to handlers.

The example shown above can use `EventHandler` delegate without declaring a custom `Notify` delegate, as shown below.

Example: EventHandler

```
class Program
{
    public static void Main()
    {
        ProcessBusinessLogic bl = new ProcessBusinessLogic();
        bl.ProcessCompleted += bl_ProcessCompleted; // register with an event
        bl.StartProcess();
    }

    // event handler
    public static void bl_ProcessCompleted(object sender, EventArgs e)
    {
        Console.WriteLine("Process Completed!");
    }
}

public class ProcessBusinessLogic
{
    // declaring an event using built-in EventHandler
    public event EventHandler ProcessCompleted;

    public void StartProcess()
    {
        Console.WriteLine("Process Started!");
        // some code here..
        OnProcessCompleted(EventArgs.Empty); //No event data
    }

    protected virtual void OnProcessCompleted(EventArgs e)
    {
        ProcessCompleted?.Invoke(this, e);
    }
}
```

Try it

In the above example, the event handler `bl_ProcessCompleted()` method includes two parameters that match with EventHandler delegate. Also, passing `this` as a sender and `EventArgs.Empty`, when we raise an event using `Invoke()` in the `OnProcessCompleted()` method. Because we don't need any data for our event, it just notifies subscribers about the completion of the process, and so we passed `EventArgs.Empty`.

## Passing an Event Data

Most events send some data to the subscribers. The EventArgs class is the base class for all the event data classes. .NET includes many built-in event data classes such as SerialDataReceivedEventArgs. It follows a naming pattern of ending all event data classes with EventArgs. You can create your custom class for event data by deriving EventArgs class.

Use EventHandler<TEventArgs> to pass data to the handler, as shown below.

Example: Passing Event Data

```
class Program
{
    public static void Main()
    {
        ProcessBusinessLogic bl = new ProcessBusinessLogic();
        bl.ProcessCompleted += bl_ProcessCompleted; // register with an event
        bl.StartProcess();
    }

    // event handler
    public static void bl_ProcessCompleted(object sender, bool IsSuccessful)
    {
        Console.WriteLine("Process " + (IsSuccessful? "Completed Successfully": "failed"));
    }
}

public class ProcessBusinessLogic
{
    // declaring an event using built-in EventHandler
    public event EventHandler<bool> ProcessCompleted;

    public void StartProcess()
    {
        try
        {
            Console.WriteLine("Process Started!");

            // some code here..

            OnProcessCompleted(true);
        }
        catch(Exception ex)
        {
            OnProcessCompleted(false);
        }
    }

    protected virtual void OnProcessCompleted(bool IsSuccessful)
    {
        ProcessCompleted?.Invoke(this, IsSuccessful);
    }
}
```

Try it

In the above example, we are passing a single boolean value to the handlers that indicate whether the process completed successfully or not.

If you want to pass more than one value as event data, then create a class deriving from the EventArgs base class, as shown below.

### Example: Custom EventArgs Class

```
class ProcessEventArgs : EventArgs
{
    public bool IsSuccessful { get; set; }
    public DateTime CompletionTime { get; set; }
}
```

The following example shows how to pass custom `ProcessEventArgs` class to the handlers.

### Example: Passing Custom EventArgs

```
class Program
{
    public static void Main()
    {
        ProcessBusinessLogic bl = new ProcessBusinessLogic();
        bl.ProcessCompleted += bl_ProcessCompleted; // register with an event
        bl.StartProcess();
    }

    // event handler
    public static void bl_ProcessCompleted(object sender, ProcessEventArgs e)
    {
        Console.WriteLine("Process " + (e.IsSuccessful? "Completed Successfully": "failed"));
        Console.WriteLine("Completion Time: " + e.CompletionTime.ToLongDateString());
    }
}

public class ProcessBusinessLogic
{
    // declaring an event using built-in EventHandler
    public event EventHandler<ProcessEventArgs> ProcessCompleted;

    public void StartProcess()
    {
        var data = new ProcessEventArgs();

        try
        {
            Console.WriteLine("Process Started!");

            // some code here..

            data.IsSuccessful = true;
            data.CompletionTime = DateTime.Now;
            OnProcessCompleted(data);
        }
        catch(Exception ex)
        {
            data.IsSuccessful = false;
            data.CompletionTime = DateTime.Now;
            OnProcessCompleted(data);
        }
    }

    protected virtual void OnProcessCompleted(ProcessEventArgs e)
    {
        ProcessCompleted?.Invoke(this, e);
    }
}
```

Try it

Thus, you can create, raise, register, and handle events in C#.

Learn What is the difference between delegate and event in C#?.

Points to Remember :

1) An event is a wrapper around a delegate. It depends on the delegate.

2) Use "event" keyword with delegate type variable to declare an event.

3) Use built-in delegate EventHandler or EventHandler<TEventArgs> for common events.

4) The publisher class raises an event, and the subscriber class registers for an event and provides the event-handler method.

5) Name the method which raises an event prefixed with "On" with the event name.

6) The signature of the handler method must match the delegate signature.

7) Register with an event using the += operator. Unsubscribe it using the -= operator. Cannot use the = operator.

8) Pass event data using EventHandler<TEventArgs>.

9) Derive EventArgs base class to create custom event data class.

10) Events can be declared static, virtual, sealed, and abstract.

11) An Interface can include the event as a member.

12) Event handlers are invoked synchronously if there are multiple subscribers.

C# Questions & Answers

Start C# Skill Test

Related Articles

- Difference between delegates and events in C#
- Difference between Array and ArrayList
- Difference between Hashtable and Dictionary
- How to write file using StreamWriter in C#?
- How to sort the generic SortedList in the descending order?
- How to read file using StreamReader in C#?
- How to calculate the code execution time in C#?
- Design Principle vs Design Pattern
- How to convert string to int in C#?
- Boxing and Unboxing in C#
- More C# articles

TutorialsTeacher
Author

We are a team of passionate developers, educators, and technology enthusiasts who, with their combined expertise and experience, create in -depth, comprehensive, and easy to understand tutorials.We focus on a blend of theoretical explanations and practical examples to encourages hands - on learning. Learn more about us .



Previous Next

## Programming Tutorials



### TutorialsTeacher.com



Contact Us

### E-mail list

Subscribe to TutorialsTeacher email list and get latest updates, tips & tricks on C#, .Net, JavaScript, jQuery, AngularJS, Node.js to your inbox.

Email address GO

We respect your privacy.





© 2024 TutorialsTeacher.com. All Rights Reserved.