F# - EVENTS

Events allow classes to send and receive messages between one another.

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

Objects communicate with one another through synchronous message passing.

Events are attached to other functions; objects register **callback** functions to an event, and these callbacks are executed when $andif$ the event is triggered by some object.

The Event Class and Event Module

The Control.Event<'T> Class helps in creating an observable object or event.

It has the following instance members to work with the events −

Member	Description
Publish	Publishes an observation as a first class value.
Trigger	Triggers an observation using the given parameters.

The Control.Event Module provides functions for managing event streams −

Value	Description
add : $'T \rightarrow unit$ → Event<'Del,'T> → unit	Runs the given function each time the given event is triggered.
choose : $'T \rightarrow 'Uoption$ → IEvent<'Del,'T> → IEvent<'U>	Returns a new event which fires on a selection of messages from the original event. The selection function takes an original message to an optional new message.
filter : $'T \rightarrow bool$ → IEvent<'Del,'T> → IEvent<'T>	Returns a new event that listens to the original event and triggers the resulting event only when the argument to the event passes the given function.
map : $'T \rightarrow 'U$ → IEvent<'Del, 'T> → IEvent<'U>	Returns a new event that passes values transformed by the given function.
merge : IEvent<'Del1,'T> → IEvent<'Del2,'T> → IEvent<'T>	Fires the output event when either of the input events fire.
pairwise : IEvent<'Del,'T> → IEvent<'T * 'T>	Returns a new event that triggers on the second and subsequent triggering of the input event. The **Nth** triggering of the input event passes the arguments from the **N-1th** and **Nth** triggering as a pair. The argument passed to the **N-1th** triggering is held in hidden internal state until the **Nth** triggering occurs.
partition : $'T \rightarrow bool$ → IEvent<'Del,'T> → IEvent<'T> * IEvent<'T>	Returns a new event that listens to the original event and triggers the first resulting event if the application of the predicate to the event

	arguments returned true, and the second event if it returned false.
scan : $'U \rightarrow 'T \rightarrow 'U$ → 'U → IEvent<'Del,'T> → IEvent<'U>	Returns a new event consisting of the results of applying the given accumulating function to successive values triggered on the input event. An item of internal state records the current value of the state parameter. The internal state is not locked during the execution of the accumulation function, so care should be taken that the input IEvent not triggered by multiple threads simultaneously.
split : $'T \rightarrow Choice <'U1,'U2>$ → IEvent<'Del,'T> → IEvent<'U1> * IEvent<'U2>	Returns a new event that listens to the original event and triggers the first resulting event if the application of the function to the event arguments returned a Choice1Of2, and the second event if it returns a Choice2Of2.

Creating Events

Events are created and used through the **Event** class. The Event constructor is used for creating an event.

Example

```
type Worker(name : string, shift : string) =
    let mutable _name = name;
    let mutable _shift = shift;
    let nameChanged = new Event<unit>() (* creates event *)
    let shiftChanged = new Event<unit>() (* creates event *)

    member this.Name
        with get() = _name
        and set(value) = _name <- value

    member this.Shift
        with get() = _shift
        and set(value) = _shift <- value
```

After this you need to expose the nameChanged field as a public member, so that the listeners can hook onto the event for which, you use the **Publish** property of the event −

```
type Worker(name : string, shift : string) =
    let mutable _name = name;
    let mutable _shift = shift;

    let nameChanged = new Event<unit>() (* creates event *)
    let shiftChanged = new Event<unit>() (* creates event *)

    member this.NameChanged = nameChanged.Publish (* exposed event handler *)
    member this.ShiftChanged = shiftChanged.Publish (* exposed event handler *)

    member this.Name
        with get() = _name
        and set(value) = _name <- value
        nameChanged.Trigger() (* invokes event handler *)

    member this.Shift
        with get() = _shift
        and set(value) = _shift <- value
    shiftChanged.Trigger() (* invokes event handler *)
```

Next, you add callbacks to event handlers. Each event handler has the type IEvent<'T>, which provides several methods −

Method	Description
val Add : event:$'T \to unit$ → unit	Connects a listener function to the event. The listener will be invoked when the event is fired.
val AddHandler : 'del → unit	Connects a handler delegate object to the event. A handler can be later removed using RemoveHandler. The listener will be invoked when the event is fired.
val RemoveHandler : 'del → unit	Removes a listener delegate from an event listener store.

The following section provides a complete example.

Example

The following example demonstrates the concept and techniques discussed above −

```
type Worker(name : string, shift : string) =
   let mutable _name = name;
   let mutable _shift = shift;

   let nameChanged = new Event<unit>() (* creates event *)
   let shiftChanged = new Event<unit>() (* creates event *)

   member this.NameChanged = nameChanged.Publish (* exposed event handler *)
   member this.ShiftChanged = shiftChanged.Publish (* exposed event handler *)

   member this.Name
      with get() = _name
      and set(value) =
         _name <- value
         nameChanged.Trigger() (* invokes event handler *)

   member this.Shift
      with get() = _shift
      and set(value) =
         _shift <- value
         shiftChanged.Trigger() (* invokes event handler *)

let wk = new Worker("Wilson", "Evening")
wk.NameChanged.Add(fun () -> printfn "Worker changed name! New name: %s" wk.Name)
wk.Name <- "William"
wk.NameChanged.Add(fun () -> printfn "-- Another handler attached to NameChanged!")
wk.Name <- "Bill"

wk.ShiftChanged.Add(fun () -> printfn "Worker changed shift! New shift: %s" wk.Shift)
wk.Shift <- "Morning"
wk.ShiftChanged.Add(fun () -> printfn "-- Another handler attached to ShiftChanged!")
wk.Shift <- "Night"
```

When you compile and execute the program, it yields the following output −

```
Worker changed name! New name: William
Worker changed name! New name: Bill
-- Another handler attached to NameChanged!
Worker changed shift! New shift: Morning
Worker changed shift! New shift: Night
-- Another handler attached to ShiftChanged!
```