Events

Inhaltsverzeichnis

1 Event

Ein Event ist nichts anderes, als die allgemeine Bereitstellung eines Delegates.
Kurze Wiederholung:

```
public delegate void LogDelegate(string msg);
public class Logger
{
  private LogDelegate _messageLogger;

  public LogDelegate MessageLogger
  {
    private get => _messageLogger;
    set { _messageLogger = value; }
  }

  public void Log(string msg)
  {
    MessageLogger(msg);
  }
```

```
private Logger _logger = new();
public MainWindow() => InitializeComponent();

private void Window_Loaded(object sender, RoutedEventArgs e)
{
  _logger.MessageLogger = x => Title = $"Last Log: {x}";
}
```

Mit Event
Man stellt Delegates nämlich nicht als Variable oder Property (mit get/set) zur Verfügung, sondern mit dem Schlüsselwort **`event`**.

```
public class Logger
{
  public event LogDelegate MessageLogger;
```

Eine Verwendung in der Datenklasse könnte daher so aussehen:

```
public void Log(string msg)
{
  //log(msg);
  // window.ShowMsgInList(msg);
  if (MessageLogger != null) MessageLogger(msg);
}
```

bzw.

```
public void Log(string msg)
{
  //log(msg);
  // window.ShowMsgInList(msg);
  MessageLogger?.Invoke(msg);
}
```

Hinweis: die Abfrage auf **`null`** ist notwendig, will das Event-Objekt null ist, wenn sich kein Listener darauf registriert.

1.1 Registrieren mit +=

Auf das Event kann man dann mit **+=** sog. **Listener** (also Funktionen, die dem Delegate-Typ entsprechen) registrieren. Ein Event ist immer automatisch ein „multicast delegate“, d.h. es kann mehr als einen Listener verwalten.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  logger = new Logger();
  logger.MessageLogger += ShowMsgInList;
  logger.MessageLogger += ShowMsgIConsole;
}
```

1.1.1 Deregistrieren mit -=

Entsprechend kann man mit **-=** wieder einen Listener vom Event abmelden.

1.2 Schlüsselwort event

Das Schlüsselwort **event** ist eigentlich nur notwendig, damit man den Benutzer dazu zwingt, sich mit **+=** zur Verteilerliste hinzuzufügen und nicht zulässt, sich als einziger Listener mit **=** zu registrieren.

```
logger.MessageLogger = ShowMsgIConsole;
```

LogDelegate Logger.MessageLogger

CS0070: The event 'Logger.MessageLogger' can only appear on the left hand side of += or -= (except when used from within the type 'Logger')

Das kann man überprüfen, indem man das Schlüsselwort versuchshalber entfernt, also:

```
public class Logger
{
  public LogDelegate MessageLogger;
}
```

event fehlt

Jetzt würde folgender Code funktionieren:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  logger = new Logger();
  logger.MessageLogger += ShowMsgInList;
  logger.MessageLogger = ShowMsgIConsole;
}
```

Jedoch würde damit die Zuweisung von **`ShowMsgInConsole`** alle anderen Listener ersetzen!

1.3 Konvention für Signatur

Angenommen, man möchte jetzt das Delegate derart ändern, dass nicht nur der Message-String sondern auch die Uhrzeit als Parameter übergeben wird. In diesem Fall müsste man alle Funktionen, die mit diesem Event verwendet werden, mitändern.
Das kann man vermeiden, indem man als Parameter immer ein Objekt übergibt, das seinerseits Properties für die verschiedenen Werte zur Verfügung stellt. Kommt dann später eine Property dazu, ändert sich die Signatur nicht mehr!
Das ist auch die Konvention, die für Events vorgeschrieben ist (und die sich alle Events in WPF halten):

- Klasse von **`EventArgs`** ableiten und darin die gewünschten Properties notieren. Der Name der Klasse endet auf „EventArgs“.
- Als zusätzlichen Parameter gibt man ein **`object`** mit, das den Sender repräsentiert (praktisch immer this). Damit hätte der Listener noch die Möglichkeit, etwaige Properties abzufragen, die nicht automatisch mitgeschickt werden.

Hinweis: beim Observer-Pattern gibt es die Variante **push** (Werte ungefragt übergeben) und **pull** (die Werte holen lassen). Der EventArgs-Parameter repräsentiert den push-Teil, der object-Parameter den pull-Teil.
Also:

```
public class MessageEventArgs : EventArgs
{
  public string Message { get; set; }
  public DateTime Timestamp { get; set; }
}
```

Das Delegate ändert sich damit zu:

```
public delegate void LogDelegate(object sender, MessageEventArgs e);
```

Entsprechend auch die Verwendung:

```
public void Log(string msg)
{
  MessageLogger?.Invoke(this,
    new MessageEventArgs { Message = msg, Timestamp = DateTime.Now });
}
```

Auch die Registrierung ändert sich leicht:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  logger = new Logger();
  logger.MessageLogger += ShowMsgInList;
}
```

```
public void ShowMsgInList(object sender, MessageEventArgs e)
{
  lstMessages.Items.Add($"{e.Timestamp:HH:mm:ss.fff} --> {e.Message}");
}
```

1.4 EventHandler<T>

C# stellt einige vorgefertigte Delegates zur Verfügung. Damit kann man sich praktisch immer die Definition eines eigenen delegate-Typs ersparen.
Der Default Event Callback-Handler mit **eigener EventArgs**-Klasse sieht so aus:
`EventHandler<MyEventArgs>` entspricht also:

```
public delegate void EventHandler(object sender, MyEventArgs args)
```

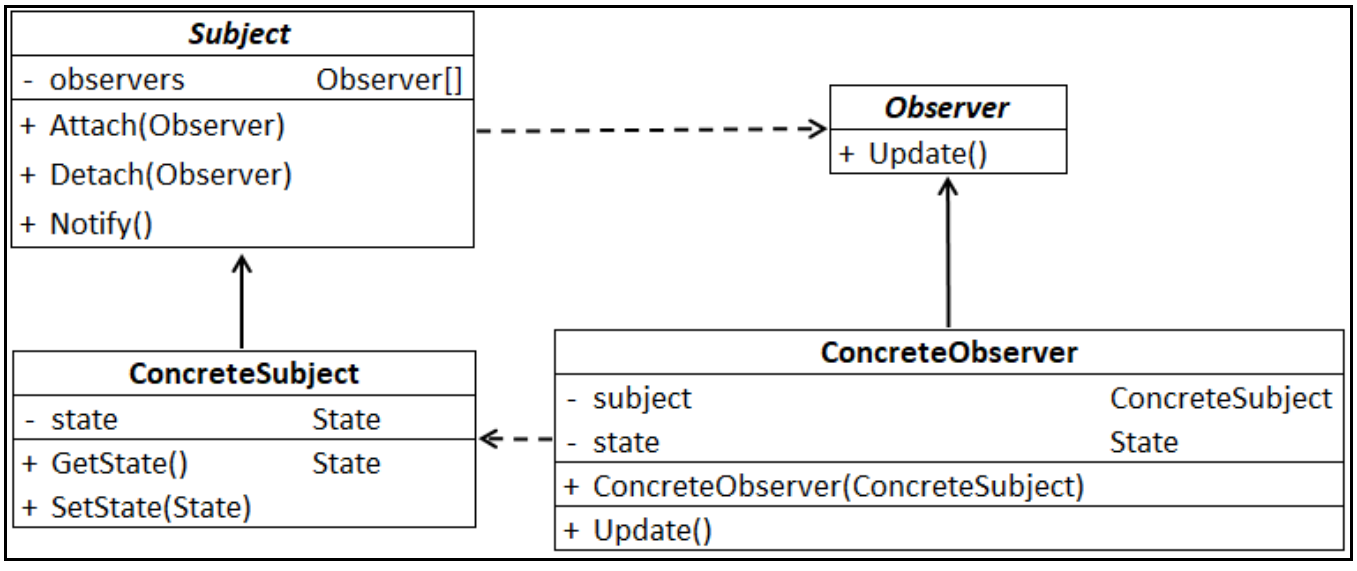
1.4.1 Beispiel

In unserem Beispiel kann man damit die eigene Definition des Delegates löschen und für das Event einen EventHandler verwenden:

```
public class Logger
{
  //public event LogDelegate MessageLogger;
  public event EventHandler<MessageEventArgs> MessageLogger;
```

1.5 Vergleich mit Observer-Pattern

In C# ist der event-Mechanismus die Umsetzung des Observer-Patterns.
Zur Erinnerung das UML-Diagramm:



Subject	Klasse	public event EventHandler<T>
	observers	Nicht explizit sichtbar – wird von einem Multicast delegate automatisch verwaltet
	Attach	+=
	Detach	-=
	Notify	Invoke
Observer	Update	Methode/Lambda-Ausdruck, den man auf das Event zuweist Update hat zwei Parameter: 1. Der erste ist der sender = das ConcreteSubject 2. Im zweiten Parameter werden bereits die wichtigsten Daten (=State) des ConcreteSubjects mitgeliefert