EVENTS WITH C# 6

Events

Events are based on delegates and offer a publish/subscribe mechanism to delegates. You can find events everywhere across the framework. In Windows applications, the `Button` class offers the `Click` event. This type of event is a delegate. A handler method that is invoked when the `Click` event is fired needs to be defined, with the parameters as defined by the delegate type.
In the code example shown in this section, events are used to connect the `CarDealer` and `Consumer` classes. The `CarDealer` class offers an event when a new car arrives. The `Consumer` class subscribes to the event to be informed when a new car arrives.

Event Publisher

You start with a `CarDealer` class that offers a subscription based on events. `CarDealer` defines the event named `NewCarInfo` of type `EventHandler<CarInfoEventArgs>` with the `event` keyword. Inside the method `NewCar`, the event `NewCarInfo` is fired by invoking the method `RaiseNewCarInfo`. The implementation of this method verifies whether the delegate is not null and raises the event (code file `EventsSample/CarDealer.cs`):

```
using static System.Console;

using System;


namespace Wrox.ProCSharp.Delegates

{

  public class CarInfoEventArgs: EventArgs

  {

    public CarInfoEventArgs(string car)

    {

       Car = car;

    }


    public string Car { get; }

  }
```

```
  public class CarDealer
  {
    public event EventHandler<CarInfoEventArgs> NewCarInfo;


    public void NewCar(string car)
    {
      WriteLine($"CarDealer, new car {car}");


      NewCarInfo?.Invoke(this, new CarInfoEventArgs(car));
    }
  }
}
```

NOTE *The null propagation operator* `.?` *used in the previous example is new with* C# 6. *This operator is discussed in Chapter 8, "Operators and Casts."*

The class `CarDealer` offers the event `NewCarInfo` of type `EventHandler<CarInfoEventArgs>`. As a convention, events typically use methods with two parameters; the first parameter is an object and contains the sender of the event, and the second parameter provides information about the event. The second parameter is different for various event types. .NET 1.0 defined several hundred delegates for events for all different data types. That's no longer necessary with the generic delegate `EventHandler<T>`. `EventHandler<TEventArgs>` defines a handler that returns `void` and accepts two parameters. With `EventHandler<TEventArgs>`, the first parameter needs to be of type `object`, and the second parameter is of type `T`. `EventHandler<TEventArgs>` also defines a constraint on `T;` it must derive from the base class `EventArgs`, which is the case with `CarInfoEventArgs`:

```
public event EventHandler<CarInfoEventArgs> NewCarInfo;
```

The delegate `EventHandler<TEventArgs>` is defined as follows:

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e)
      where TEventArgs: EventArgs
```

Defining the event in one line is a C# shorthand notation. The compiler creates a variable of the delegate type `EventHandler<CarInfoEventArgs>` and adds methods to subscribe and unsubscribe from the delegate. The long form of the shorthand notation is shown next. This is very similar to auto-properties and full properties. With events, the `add` and `remove` keywords are used to add and remove a handler to the delegate:

```
private EventHandler<CarInfoEventArgs> newCarInfo;

public event EventHandler<CarInfoEventArgs> NewCarInfo

{

  add

  {

    newCarInfo += value;

  }

  remove

  {

    newCarInfo -= value;

  }

}
```

NOTE *The long notation to define events is useful if more needs to be done than just adding and removing the event handler, such as adding synchronization for multiple thread access. The WPF controls make use of the long notation to add bubbling and tunneling functionality with the events. You can read more about event bubbling and tunneling events in Chapter 29, "Core XAML."*

The class `CarDealer` fires the event by calling the `Invoke` method of the delegate. This invokes all the handlers that are subscribed to the event. Remember, as previously shown with multicast delegates, the order of the methods invoked is not guaranteed. To have more control over calling the handler methods you can use the `Delegate` class method `GetInvocationList` to access every item in the delegate list and invoke each on its own, as shown earlier.

```
NewCarInfo?.Invoke(this, new CarInfoEventArgs(car));
```

Firing the event is just a one-liner. However, this is only with C# 6. Previous to C# 6, firing the event was more complex. Here is the same functionality implemented before C# 6. Before firing the event, you need to check whether the event is null. Because between a null check and firing the event the event could be set to null by another thread, a local variable is used, as shown in the following example:

```
EventHandler<CarInfoEventArgs> newCarInfo = NewCarInfo;

if (newCarInfo != null)

{

  newCarInfo(this, new CarInfoEventArgs(car));
```

```
}
```

With C# 6, all this could be replaced by using null propagation, with a single code line as you've seen earlier.

Before firing the event, it is necessary to check whether the delegate `NewCarInfo` is not `null`. If no one subscribed, the delegate is `null`:

```
protected virtual void RaiseNewCarInfo(string car)

{

  NewCarInfo?.Invoke(this, new CarInfoEventArgs(car));

}
```

Event Listener

The class `Consumer` is used as the event listener. This class subscribes to the event of the `CarDealer` and defines the method `NewCarIsHere` that in turn fulfills the requirements of the `EventHandler<CarInfoEventArgs>` delegate with parameters of type `object` and `CarInfoEventArgs` (code file `EventsSample/Consumer.cs`):

```
using static System.Console;


namespace Wrox.ProCSharp.Delegates

{

  public class Consumer

  {

    private string _name;


    public Consumer(string name)

    {

      _name = name;

    }


    public void NewCarIsHere(object sender, CarInfoEventArgs e)

    {

      WriteLine($"{_name}: car {e.Car} is new");
```

```
    }
  }
}
```

Now the event publisher and subscriber need to connect. This is done by using the `NewCarInfo` event of the `CarDealer` to create a subscription with `+=`. The consumer *Michael* subscribes to the event, then the consumer *Sebastian*, and next *Michael* unsubscribes with `-=` (code file `EventsSample/Program.cs`):

```
namespace Wrox.ProCSharp.Delegates
{
  class Program
  {
    static void Main()
    {
      var dealer = new CarDealer();

      var daniel = new Consumer("Daniel");
      dealer.NewCarInfo += michael.NewCarIsHere;

      dealer.NewCar("Mercedes");

      var sebastian = new Consumer("Sebastian");
      dealer.NewCarInfo += sebastian.NewCarIsHere;

      dealer.NewCar("Ferrari");

      dealer.NewCarInfo -= sebastian.NewCarIsHere;

      dealer.NewCar("Red Bull Racing");
    }
```

```
  }
}
```

Running the application, a Mercedes arrived and Daniel was informed. After that, Sebastian registers for the subscription as well, both Daniel and Sebastian are informed about the new Ferrari. Then Sebastian unsubscribes and only Daniel is informed about the Red Bull:

```
CarDealer, new car Mercedes

Daniel: car Mercedes is new

CarDealer, new car Ferrari

Daniel: car Ferrari is new

Sebastian: car Ferrari is new

CarDealer, new car Red Bull Racing

Daniel: car Red Bull is new
```