CHAPTER 11

Events

In this chapter:

In this chapter, I'll talk about the last kind of member a type can define: events. A type that defines an event member allows the type (or instances of the type) to notify other objects that something special has happened. For example, the `Button` class offers an event called `Click`. When a `Button` object is clicked, one or more objects in an application may want to receive notification about this event in order to perform some action. Events are type members that allow this interaction. Specifically, defining an event member means that a type is offering the following capabilities:

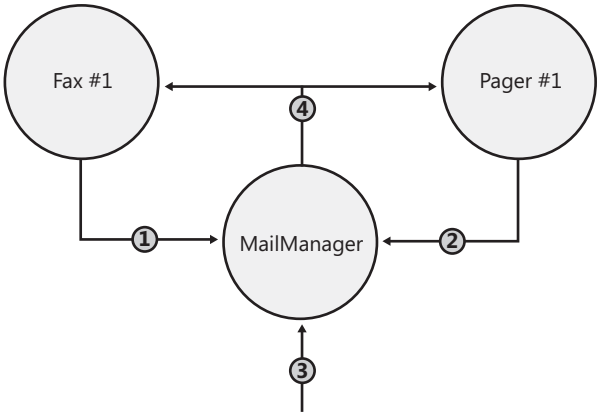
- A method can register its interest in the event.
- A method can unregister its interest in the event.
- Registered methods will be notified when the event occurs.

Types can offer this functionality when defining an event because they maintain a list of the registered methods. When the event occurs, the type notifies all of the registered methods in the collection.

The common language runtime's (CLR's) event model is based on *delegates*. A delegate is a type-safe way to invoke a callback method. Callback methods are the means by which objects receive the notifications they subscribed to. In this chapter, I'll be using delegates, but I won't fully explain all their details until Chapter 17, "Delegates."

To help you fully understand the way events work within the CLR, I'll start with a scenario in which events are useful. Suppose you want to design an email application. When an email message arrives, the user might like the message to be forwarded to a fax machine or a pager. In architecting this application, let's say that you'll first design a type called `MailManager` that receives the incoming email messages. `MailManager` will expose an event called `NewMail`. Other types (such as `Fax` and `Pager`) may register interest in this event. When `MailManager` receives a new email message, it will raise the event, causing the message to be distributed to each of the registered objects. Each object can process the message in any way it desires.

When the application initializes, let's instantiate just one `MailManager` instance—the application can then instantiate any number of `Fax` and `Pager` types. Figure 11-1 shows how the application initializes and what happens when a new email message arrives.



1. A method in the Fax object registers interest with the MailManager's event.
2. A method in the Pager object registers interest with the MailManager's event.
3. A new mail message arrives at MailManager.
4. The MailManager object fires the notification off to all the registered methods, which process the mail message as desired.

FIGURE 11-1 Architecting an application to use events.

Here's how the application illustrated in Figure 11-1 works: the application initializes by constructing an instance of `MailManager`. `MailManager` offers a `NewMail` event. When the `Fax` and `Pager` objects are constructed, they register an instance method with `MailManager`'s `NewMail` event so that `MailManager` knows to notify the `Fax` and `Pager` objects when new email messages arrive. Now, when `MailManager` receives a new email message (sometime in the future), it will raise the `NewMail` event, giving all of the registered methods an opportunity to process the new message in any way they want.

Designing a Type That Exposes an Event

There are many steps a developer must take in order to define a type that exposes one or more event members. In this section, I'll walk through each of the necessary steps. The `MailManager` sample application (which can be downloaded from the Books section in Resources at *http://wintellect.com/Books*) shows all of the source code for the `MailManager` type, the `Fax` type, and the `Pager` type. You'll notice that the `Pager` type is practically identical to the `Fax` type.

Step #1: Define a type that will hold any additional information that should be sent to receivers of the event notification

When an event is raised, the object raising the event may want to pass some additional information to the objects receiving the event notification. This additional information needs to be encapsulated into its own class, which typically contains a bunch of private fields along with some read-only public properties to expose these fields. By convention, classes that hold event information to be passed to the event handler should be derived from `System.EventArgs`, and the name of the class should be suffixed with `EventArgs`. In this example, the `NewMailEventArgs` class has fields identifying who sent the message (`m_from`), who is receiving the message (`m_to`), and the subject of the message (`m_subject`).

```
// Step #1: Define a type that will hold any additional information that
// should be sent to receivers of the event notification
internal class NewMailEventArgs : EventArgs {

   private readonly String m_from, m_to, m_subject;

   public NewMailEventArgs(String from, String to, String subject) {
      m_from = from; m_to = to; m_subject = subject;
   }

   public String From    { get { return m_from;    } }
   public String To      { get { return m_to;      } }
   public String Subject { get { return m_subject; } }
}
```

Note The `EventArgs` class is defined in the Microsoft .NET Framework Class Library (FCL) and is implemented like the following.

```
[ComVisible(true), Serializable]
public class EventArgs {
   public static readonly EventArgs Empty = new EventArgs();
   public EventArgs() {  }
}
```

As you can see, this type is nothing to write home about. It simply serves as a base type from which other types can derive. Many events don't have any additional information to pass on. For example, when a `Button` notifies its registered receivers that it has been clicked, just invoking the callback method is enough information. When you're defining an event that doesn't have any additional data to pass on, just use `EventArgs.Empty` rather than constructing a new `EventArgs` object.

Step #2: Define the event member

An event member is defined using the C# keyword `event`. Each event member is given accessibility (which is almost always `public` so that other code can access the event member), a type of delegate indicating the prototype of the method(s) that will be called, and a name (which can be any valid identifier). Here is what the event member in our `MailManager` class looks like.

```
internal class MailManager {

   // Step #2: Define the event member
   public event EventHandler<NewMailEventArgs> NewMail;
   ...
}
```

`NewMail` is the name of this event. The type of the event member is `EventHandler<NewMailEventArgs>`, which means that all receivers of the event notification must supply a callback method whose prototype matches that of the `EventHandler<NewMailEventArgs>` delegate type. Because the generic `System.EventHandler` delegate is defined as follows.

```
public delegate void EventHandler<TEventArgs>(Object sender, TEventArgs e);
```

the method prototypes must look like the following.

```
void MethodName(Object sender, NewMailEventArgs e);
```

Note A lot of people wonder why the event pattern requires the `sender` parameter to always be of type `Object`. After all, because the `MailManager` will be the only type raising an event with a `NewMailEventArgs` object, it makes more sense for the callback method to be prototyped like the following.

```
void MethodName(MailManager sender, NewMailEventArgs e);
```

The pattern requires the sender parameter to be of type `Object` mostly because of inheritance. What if `MailManager` were used as a base class for `SmtpMailManager`? In this case, the callback method should have the `sender` parameter prototyped as `SmtpMailManager` instead of `MailManager`, but this can't happen because `SmtpMailManager` just inherited the `NewMail` event. So the code that was expecting `SmtpMailManager` to raise the event must still have to cast the `sender` argument to `SmtpMailManager`. In other words, the cast is still required, so the sender parameter might as well be typed as `Object`.

The next reason for typing the `sender` parameter as `Object` is just flexibility. It allows the delegate to be used by multiple types that offer an event that passes a `NewMailEventArgs` object. For example, a `PopMailManager` class could use the delegate even if this class were not derived from `MailManager`.

The event pattern also requires that the delegate definition and the callback method name the `EventArgs`-derived parameter e. The only reason for this is to add additional consistency to the pattern, making it easier for developers to learn and implement the pattern. Tools that spit out source code (such as Microsoft Visual Studio) also know to call the parameter e.

Finally, the event pattern requires all event handlers to have a return type of `void`. This is necessary because raising an event might call several callback methods, and there is no way to get the return values from all of them. Having a return type of `void` doesn't allow the callbacks to return a value. Unfortunately, there are some event handlers in the FCL, such as `ResolveEventHandler`, that did not follow Microsoft's own prescribed pattern because it returns an object of type `Assembly`.

Step #3: Define a method responsible for raising the event to notify registered objects that the event has occurred

By convention, the class should define a protected, virtual method that is called by code internally within the class and its derived classes when the event is to be raised. This method takes one parameter, a `NewMailEventArgs` object, which includes the information passed to the objects receiving the notification. The default implementation of this method simply checks if any objects have registered interest in the event and, if so, the event will be raised, thereby notifying the registered methods that the event has occurred. Here is what the method in our `MailManager` class looks like.

```
internal class MailManager {
   ...
   // Step #3: Define a method responsible for raising the event
   // to notify registered objects that the event has occurred
   // If this class is sealed, make this method private and nonvirtual
   protected virtual void OnNewMail(NewMailEventArgs e) {

      // Copy a reference to the delegate field now into a temporary field for thread safety
      EventHandler<NewMailEventArgs> temp = Volatile.Read(ref NewMail);

      // If any methods registered interest with our event, notify them
      if (temp != null) temp(this, e);
   }
   ...
}
```

Raising an Event in a Thread-Safe Way

When the .NET Framework first shipped, the recommended way for developers to raise an event was by using code similar to the following.

```
// Version 1
protected virtual void OnNewMail(NewMailEventArgs e) {
   if (NewMail != null) NewMail(this, e);
}
```

The problem with the `OnNewMail` method is that the thread could see that `NewMail` is not `null`, and then, just before invoking `NewMail`, another thread could remove a delegate from the chain making `NewMail` `null`, resulting in a `NullReferenceException` being thrown. To fix this race condition, many developers write the `OnNewMail` method as follows.

```
// Version 2
protected virtual void OnNewMail(NewMailEventArgs e) {
   EventHandler<NewMailEventArgs> temp = NewMail;
   if (temp != null) temp(this, e);
}
```

The thinking here is that a reference to `NewMail` is copied into a temporary variable, `temp`, which refers to the chain of delegates at the moment the assignment is performed. Now, this method compares `temp` and `null` and invokes `temp`, so it doesn't matter if another thread changes `NewMail` after the assignment to `temp`. Remember that delegates are immutable and this is why this technique works in theory. However, what a lot of developers don't realize is that this code could be optimized by the compiler to remove the local `temp` variable entirely. If this happens, this version of the code is identical to the first version, so a `NullReferenceException` is still possible.

To really fix this code, you should rewrite `OnNewMail` like the following.

```
// Version 3
protected virtual void OnNewMail(NewMailEventArgs e) {
   EventHandler<NewMailEventArgs> temp = Volatile.Read(ref NewMail);
   if (temp != null) temp(this, e);
}
```

The call to `Volatile.Read` forces `NewMail` to be read at the point of the call and the reference really has to be copied to the `temp` variable now. Then, `temp` will be invoked only if it is not `null`. See Chapter 29, "Primitive Thread Synchronization Constructs," for more information about the `Volatile.Read` method.

Although the last version of this code is the best, technically correct version, you can actually use the second version because the just-in-time (JIT) compiler is aware of this pattern and it knows not to optimize away the local `temp` variable. Specifically, all of Microsoft's JIT compilers respect the invariant of not introducing new reads to heap memory and therefore, caching a reference in a local variable ensures that the heap reference is accessed only once. This is not documented and, in theory, it could change, which is why you should use the last version. But in reality, Microsoft's JIT compiler would never embrace a change that would break this pattern because too many applications would break.[1] In addition, events are mostly used in single-threaded scenarios (Windows Presentation Foundation and Windows Store apps) and so thread safety is not an issue anyway.

It is very important to note that due to this thread race condition, it is also possible that a method will be invoked after it has been removed from the event's delegate chain.

As a convenience, you could define an extension method (as discussed in Chapter 8, "Methods") that encapsulates this thread-safety logic. Define the extension method as follows.

```
public static class EventArgExtensions {
   public static void Raise<TEventArgs>(this TEventArgs e,
      Object sender, ref EventHandler<TEventArgs> eventDelegate) {

      // Copy a reference to the delegate field now into a temporary field for thread safety
      EventHandler<TEventArgs> temp = Volatile.Read(ref eventDelegate);

      // If any methods registered interest with our event, notify them
      if (temp != null) temp(sender, e);
   }
}
```

And now, we can rewrite the `OnNewMail` method as follows.

```
protected virtual void OnNewMail(NewMailEventArgs e) {
   e.Raise(this, ref m_NewMail);
}
```

A class that uses `MailManager` as a base type is free to override the `OnNewMail` method. This capability gives the derived class control over the raising of the event. The derived class can handle the new email message in any way it sees fit. Usually, a derived type calls the base type's `OnNewMail` method so that the registered method(s) receive the notification. However, the derived class might decide to disallow the event from being forwarded.

[1] This was actually told to me by a member of the Microsoft JIT compiler team.

Step #4: Define a method that translates the input into the desired event

Your class must have some method that takes some input and translates it into the raising of the event. In my `MailManager` example, the `SimulateNewMail` method is called to indicate that a new email message has arrived into `MailManager`.

```
internal class MailManager {

   // Step #4: Define a method that translates the
   // input into the desired event
   public void SimulateNewMail(String from, String to, String subject) {

      // Construct an object to hold the information we want
      // to pass to the receivers of our notification
      NewMailEventArgs e = new NewMailEventArgs(from, to, subject);

      // Call our virtual method notifying our object that the event
      // occurred. If no type overrides this method, our object will
      // notify all the objects that registered interest in the event
      OnNewMail(e);
   }
}
```

`SimulateNewMail` accepts information about the message and constructs a `NewMailEventArgs` object, passing the message information to its constructor. `MailManager`'s own virtual `OnNewMail` method is then called to formally notify the `MailManager` object of the new email message. Usually, this causes the event to be raised, notifying all of the registered methods. (As mentioned before, a class using `MailManager` as a base class can override this behavior.)

How the Compiler Implements an Event

Now that you know how to define a class that offers an event member, let's take a closer look at what an event really is and how it works. In the `MailManager` class, we have a line of code that defines the event member itself.

```
public event EventHandler<NewMailEventArgs> NewMail;
```

When the C# compiler compiles the line above, it translates this single line of source code into the following three constructs.

```
// 1. A PRIVATE delegate field that is initialized to null
private EventHandler<NewMailEventArgs> NewMail = null;

// 2. A PUBLIC add_Xxx method (where Xxx is the Event name)
// Allows methods to register interest in the event.
public void add_NewMail(EventHandler<NewMailEventArgs> value) {
   // The loop and the call to CompareExchange is all just a fancy way
   // of adding a delegate to the event in a thread-safe way
   EventHandler<NewMailEventArgs>prevHandler;
```

```
   EventHandler<NewMailEventArgs> newMail = this.NewMail;
   do {
      prevHandler = newMail;
      EventHandler<NewMailEventArgs> newHandler =
         (EventHandler<NewMailEventArgs>) Delegate.Combine(prevHandler, value);
      newMail = Interlocked.CompareExchange<EventHandler<NewMailEventArgs>>(
         ref this.NewMail, newHandler, prevHandler);
   } while (newMail != prevHandler);
}

// 3. A PUBLIC remove_Xxx method (where Xxx is the Event name)
// Allows methods to unregister interest in the event.
public void remove_NewMail(EventHandler<NewMailEventArgs> value) {
   // The loop and the call to CompareExchange is all just a fancy way
   // of removing a delegate from the event in a thread-safe way
   EventHandler<NewMailEventArgs> prevHandler;
   EventHandler<NewMailEventArgs> newMail = this.NewMail;
   do {
      prevHandler = newMail;
      EventHandler<NewMailEventArgs> newHandler =
         (EventHandler<NewMailEventArgs>) Delegate.Remove(prevHandler, value);
      newMail = Interlocked.CompareExchange<EventHandler<NewMailEventArgs>>(
         ref this.NewMail, newHandler, prevHandler);
   } while (newMail != prevHandler);
}
```

The first construct is simply a field of the appropriate delegate type. This field is a reference to the head of a list of delegates that will be notified when this event occurs. This field is initialized to `null`, meaning that no listeners have registered interest in the event. When a method registers interest in the event, this field refers to an instance of the `EventHandler<NewMailEventArgs>` delegate, which may refer to additional `EventHandler<NewMailEventArgs>` delegates. When a listener registers interest in an event, the listener is simply adding an instance of the delegate type to the list. Obviously, unregistering means removing the delegate from the list.

You'll notice that the delegate field, `NewMail` in this example, is always `private` even though the original line of source code defines the event as `public`. The reason for making the delegate field `private` is to prevent code outside the defining class from manipulating it improperly. If the field were `public`, any code could alter the value in the field and potentially wipe out all of the delegates that have registered interest in the event.

The second construct the C# compiler generates is a method that allows other objects to register their interest in the event. The C# compiler automatically names this function by prepending `add_` to the event's name (`NewMail`). The C# compiler automatically generates the code that is inside this method. The code always calls `System.Delegate`'s static `Combine` method, which adds the instance of a delegate to the list of delegates and returns the new head of the list, which gets saved back in the field.

The third construct the C# compiler generates is a method that allows an object to unregister its interest in the event. Again, the C# compiler automatically names this function by prepending `remove_` to the event's name (`NewMail`). The code inside this method always calls `Delegate`'s static `Remove` method, which removes the instance of a delegate from the list of delegates and returns the new head of the list, which gets saved back in the field.

Warning If you attempt to remove a method that was never added, then `Delegate`'s `Remove` method internally does nothing. That is, you get no exception or warning of any type; the event's collection of methods remains unchanged.

Note The `add` and `remove` methods use a well-known pattern to update a value in a thread-safe way. This pattern is discussed in the "The Interlocked Anything Pattern" section of Chapter 29.

In this example, the `add` and `remove` methods are `public`. The reason they are `public` is that the original line of source code declared the event to be `public`. If the event had been declared `protected`, the `add` and `remove` methods generated by the compiler would also have been declared `protected`. So, when you define an event in a type, the accessibility of the event determines what code can register and unregister interest in the event, but only the type itself can ever access the delegate field directly. Event members can also be declared as `static` or `virtual`, in which case the `add` and `remove` methods generated by the compiler would be either `static` or `virtual`, respectively.

In addition to emitting the aforementioned three constructs, compilers also emit an event definition entry into the managed assembly's metadata. This entry contains some flags and the underlying delegate type, and refers to the `add` and `remove` accessor methods. This information exists simply to draw an association between the abstract concept of an "event" and its accessor methods. Compilers and other tools can use this metadata, and this information can also be obtained by using the `System.Reflection.EventInfo` class. However, the CLR itself doesn't use this metadata information and requires only the accessor methods at run time.

Designing a Type That Listens for an Event

The hard work is definitely behind you at this point. In this section, I'll show you how to define a type that uses an event provided by another type. Let's start off by examining the code for the Fax type.

```
internal sealed class Fax {
   // Pass the MailManager object to the constructor
   public Fax(MailManager mm) {

      // Construct an instance of the EventHandler<NewMailEventArgs>
      // delegate that refers to our FaxMsg callback method.
      // Register our callback with MailManager's NewMail event
      mm.NewMail += FaxMsg;
   }

   // This is the method the MailManager will call
   // when a new email message arrives
   private void FaxMsg(Object sender, NewMailEventArgs e) {
```

```
      // 'sender' identifies the MailManager object in case
      // we want to communicate back to it.

      // 'e' identifies the additional event information
      // the MailManager wants to give us.

      // Normally, the code here would fax the email message.
      // This test implementation displays the info in the console
      Console.WriteLine("Faxing mail message:");
      Console.WriteLine("   From={0}, To={1}, Subject={2}",
         e.From, e.To, e.Subject);
   }

   // This method could be executed to have the Fax object unregister
   // itself with the NewMail event so that it no longer receives
   // notifications
   public void Unregister(MailManager mm) {

      // Unregister with MailManager's NewMail event
      mm.NewMail -= FaxMsg;
   }
}
```

When the email application initializes, it would first construct a `MailManager` object and save the reference to this object in a variable. Then the application would construct a `Fax` object, passing the reference to the `MailManager` object as a parameter. In the `Fax` constructor, the `Fax` object registers its interest in `MailManager`'s `NewMail` event using C#'s += operator.

```
mm.NewMail += FaxMsg;
```

Because the C# compiler has built-in support for events, the compiler translates the use of the += operator into the following line of code to add the object's interest in the event.

```
mm.add_NewMail(new EventHandler<NewMailEventArgs>(this.FaxMsg));
```

As you can see, the C# compiler is generating code that will construct an `EventHandler<NewMailEventArgs>` delegate object that wraps the `Fax` class's `FaxMsg` method. Then, the C# compiler calls the `MailManager`'s `add_NewMail` method, passing it the new delegate. Of course, you can verify all of this by compiling the code and looking at the IL with a tool such as ILDasm.exe.

Even if you're using a programming language that doesn't directly support events, you can still register a delegate with the event by calling the `add` accessor method explicitly. The effect is identical; the source code will just not look as pretty. It's the `add` method that registers the delegate with the event by adding it to the event's list of delegates.

When the `MailManager` object raises the event, the `Fax` object's `FaxMsg` method gets called. The method is passed a reference to the `MailManager` object as the first parameter, `sender`. Most of the time, this parameter is ignored, but it can be used if the `Fax` object wants to access members of the `MailManager` object in response to the event notification. The second parameter is a reference to a `NewMailEventArgs` object. This object contains any additional information the designer of `MailManager` and `NewMailEventArgs` thought would be useful to the event receivers.

From the NewMailEventArgs object, the FaxMsg method has easy access to the message's sender, the message's recipient, and the message's subject. In a real Fax object, this information would be faxed somewhere. In this example, the information is simply displayed in the console window.

When an object is no longer interested in receiving event notifications, it should unregister its interest. For example, the Fax object would unregister its interest in the NewMail event if the user no longer wanted his or her email forwarded to a fax. As long as an object has registered one of its methods with an event, the object can't be garbage collected. If your type implements IDisposable's Dispose method, the implementation should cause it to unregister interest in all events. (See Chapter 21, "The Managed Heap and Garbage Collection," for more information about IDisposable.)

Code that demonstrates how to unregister for an event is shown in Fax's Unregister method. This method is practically identical to the code shown in the Fax constructor. The only difference is that this code uses -= instead of +=. When the C# compiler sees code using the -= operator to unregister a delegate with an event, the compiler emits a call to the event's remove method.

```
mm.remove_NewMail(new EventHandler<NewMailEventArgs>(FaxMsg));
```

As with the += operator, even if you're using a programming language that doesn't directly support events, you can still unregister a delegate with the event by calling the remove accessor method explicitly. The remove method unregisters the delegate from the event by scanning the list for a delegate that wraps the same method as the one passed in. If a match is found, the existing delegate is removed from the event's list of delegates. If a match isn't found, no error occurs, and the list is unaltered.

By the way, C# requires your code to use the += and -= operators to add and remove delegates from the list. If you try to call the add or remove method explicitly, the C# compiler produces the CS0571 cannot explicitly call operator or accessor error message.

Explicitly Implementing an Event

The System.Windows.Forms.Control type defines about 70 events. If the Control type implemented the events by allowing the compiler to implicitly generate the add and remove accessor methods and delegate fields, every Control object would have 70 delegate fields in it just for the events! Because most programmers care about just a few events, an enormous amount of memory would be wasted for each object created from a Control-derived type. By the way, the ASP.NET System.Web.UI.Control and the Windows Presentation Foundation (WPF) System.Windows.UIElement type also offer many events that most programmers do not use.

In this section, I discuss how the C# compiler allows a class developer to explicitly implement an event, allowing the developer to control how the add and remove methods manipulate the callback delegates. I'm going to demonstrate how explicitly implementing an event can be used to efficiently implement a class that offers many events. However, there are certainly other scenarios where you might want to explicitly implement a type's event.

To efficiently store event delegates, each object that exposes events will maintain a collection (usually a dictionary) with some sort of event identifier as the key and a delegate list as the value. When a new object is constructed, this collection is empty. When interest in an event is registered, the event's identifier is looked up in the collection. If the event identifier is there, the new delegate is combined with the list of delegates for this event. If the event identifier isn't in the collection, the event identifier is added with the delegate.

When the object needs to raise an event, the event identifier is looked up in the collection. If the collection doesn't have an entry for the event identifier, nothing has registered interest in the event and no delegates need to be called back. If the event identifier is in the collection, the delegate list associated with the event identifier is invoked. Implementing this design pattern is the responsibility of the developer who is designing the type that defines the events; the developer using the type has no idea how the events are implemented internally.

Here is an example of how you could accomplish this pattern. First, I implemented an `EventSet` class that represents a collection of events and each event's delegate list as follows.

```
using System;
using System.Collections.Generic;

// This class exists to provide a bit more type safety and
// code maintainability when using EventSet
public sealed class EventKey { }

public sealed class EventSet {
   // The private dictionary used to maintain EventKey -> Delegate mappings
private readonly Dictionary<EventKey, Delegate> m_events =
   new Dictionary<EventKey, Delegate>();

// Adds an EventKey -> Delegate mapping if it doesn't exist or
// combines a delegate to an existing EventKey
public void Add(EventKey eventKey, Delegate handler) {
   Monitor.Enter(m_events);
   Delegate d;
   m_events.TryGetValue(eventKey, out d);
   m_events[eventKey] = Delegate.Combine(d, handler);
   Monitor.Exit(m_events);
}

// Removes a delegate from an EventKey (if it exists) and
// removes the EventKey -> Delegate mapping the last delegate is removed
public void Remove(EventKey eventKey, Delegate handler) {
   Monitor.Enter(m_events);
   // Call TryGetValue to ensure that an exception is not thrown if
   // attempting to remove a delegate from an EventKey not in the set
   Delegate d;
   if (m_events.TryGetValue(eventKey, out d)) {
      d = Delegate.Remove(d, handler);
```

```
      // If a delegate remains, set the new head else remove the EventKey
      if (d != null) m_events[eventKey] = d;
      else m_events.Remove(eventKey);
   }
   Monitor.Exit(m_events);
}

// Raises the event for the indicated EventKey
public void Raise(EventKey eventKey, Object sender, EventArgs e) {
   // Don't throw an exception if the EventKey is not in the set
   Delegate d;
   Monitor.Enter(m_events);
   m_events.TryGetValue(eventKey, out d);
   Monitor.Exit(m_events);

   if (d != null) {
      // Because the dictionary can contain several different delegate types,
      // it is impossible to construct a type-safe call to the delegate at
      // compile time. So, I call the System.Delegate type's DynamicInvoke
      // method, passing it the callback method's parameters as an array of
      // objects. Internally, DynamicInvoke will check the type safety of the
      // parameters with the callback method being called and call the method.
      // If there is a type mismatch, then DynamicInvoke will throw an exception.
      d.DynamicInvoke(new Object[] { sender, e });
      }
   }
```

Note The FCL defines a type, `System.Windows.EventHandlersStore`, which does essentially the same thing as my `EventSet` class. Various WPF types use the `EventHandlersStore` type internally to maintain their sparse set of events. You're certainly welcome to use the FCL's `EventHandlersStore` type if you'd like. The big difference between the `EventHandlersStore` type and my `EventSet` type is that `EventHandlersStore` doesn't offer any thread-safe way to access the events; you would have to implement your own thread-safe wrapper around the `EventHandlersStore` collection if you need to do this.

Now, I show a class that uses my `EventSet` class. This class has a field that refers to an `EventSet` object, and each of this class's events is explicitly implemented so that each event's `add` method stores the specified callback delegate in the `EventSet` object and each event's `remove` method eliminates the specified callback delegate (if found).

```
using System;

// Define the EventArgs-derived type for this event.
public class FooEventArgs : EventArgs { }

public class TypeWithLotsOfEvents {

   // Define a private instance field that references a collection.
   // The collection manages a set of Event/Delegate pairs.
   // NOTE: The EventSet type is not part of the FCL, it is my own type.
   private readonly EventSet m_eventSet = new EventSet();
```

```
   // The protected property allows derived types access to the collection.
   protected EventSet EventSet { get { return m_eventSet; } }

   #region Code to support the Foo event (repeat this pattern for additional events)
   // Define the members necessary for the Foo event.
   // 2a. Construct a static, read-only object to identify this event.
   // Each object has its own hash code for looking up this
   // event's delegate linked list in the object's collection.
   protected static readonly EventKey s_fooEventKey = new EventKey();

   // 2b. Define the event's accessor methods that add/remove the
   // delegate from the collection.
   public event EventHandler<FooEventArgs> Foo {
      add    { m_eventSet.Add(s_fooEventKey, value); }
      remove { m_eventSet.Remove(s_fooEventKey, value); }

   }

   // 2c. Define the protected, virtual On method for this event.
   protected virtual void OnFoo(FooEventArgs e) {
      m_eventSet.Raise(s_fooEventKey, this, e);
   }

   // 2d. Define the method that translates input to this event.
   public void SimulateFoo() { OnFoo(new FooEventArgs()); }
   #endregion
}
```

Code that uses the `TypeWithLotsOfEvents` type can't tell whether the events have been implemented implicitly by the compiler or explicitly by the developer. They just register the events by using normal syntax. Here is some code demonstrating this.

```
public sealed class Program {
   public static void Main() {
      TypeWithLotsOfEvents twle = new TypeWithLotsOfEvents();

      // Add a callback here
      twle.Foo += HandleFooEvent;

      // Prove that it worked
      twle.SimulateFoo();
   }

   private static void HandleFooEvent(object sender, FooEventArgs e) {
      Console.WriteLine("Handling Foo Event here...");
   }
}
```