Chapter 9
Delegates and Events

Two language features are central to the use of the .NET FCL. We have been using both delegates and events in the topics we have discussed so far, but I haven’t explored the inner workings or how to create your own delegates and events. In the next few sections I will do just that.

Pointers to Functions

To help C and C++ programmers to more easily understand the concept of delegates let’s look at the use of pointers to functions in C++. This is the same technique you would use in C as well. Figure 9-1 shows a program that defines two simple functions, $A$ and $B$, that merely output a line of text echoing the integer argument.

The $main$ function declares a pointer to a function, $pf$, that is a pointer to a function taking a single integer argument and doesn't return a value. The following line is this declaration:

```cpp
void (*pf)(int i);
```

The pair of parenthesis around $*pf$ are required. If the are omitted then the declaration merely declares a function that returns a pointer to $void$. That’s not what we want. We can set $pf$ to point to either $A$ or $B$ and then use $pf$ as the function to call. Depending on whether it points to $A$ or $B$ the corresponding function will be called. The output is shown in Figure 9-2.

```cpp
#include <iostream>
using namespace std;
void A(int i)
{
    cout << "A called with " << i << endl;
}
void B(int i)
{
    cout << "B called with " << i << endl;
}
void main()
{
    void (*pf)(int i);
    pf=A;
    pf(1);
    pf=B;
    pf(2);
    return;
}
```

Figure 9-1
Pointers to functions are often used as arguments to other functions such that they can utilize some arbitrary function specified by the argument. For example, we might define some type of data management tool that requires a comparison operation. By using a pointer to a function we could pass this pointer to the data management tool as an argument and it would use whatever function you wanted. The data management tool would not have to build in the name of the comparison function. Building in the name of the function would also mean that we would have to have several versions of this tool, one for each different comparison function you wrote.

The .NET FCL actually uses this very technique with its collection classes. Of course a delegate is used since pointers are available only with unmanaged code and we want to avoid that if at all possible. Let's focus our attention on delegates and how they provide the same capabilities as pointers to functions with some added features as well.

Figure 9-2

The Delegate Equivalent

A delegate is declared by prefacing the signature for a method with the keyword delegate. A delegate can be declared with global scope or local to a class. Making it global allows it to be used by many different classes without tying it to one class in particular. This is similar to the way we usually use enumerations in the .NET FCL. The following line declares a delegate to a method taking an integer argument and not returning a value:

delegate void myDelegate(int i);

In order to use the delegate to call a method we need to create an instance of the delegate and associate it with a particular method. The method may be a static or instance method. It makes no difference. By the way, this is a very significant improvement over pointers to functions. In C++ it is tricky to use pointer to functions that are associated with the instance of a class. The following four lines show how we can create a delegate associated with method A, call it, associate a new instance of the delegate with method B, and finally call that method:
delegate void myDelegate(int i);

class Program
{
    static void A(int i)
    {
        Console.WriteLine("A called with {0}", i);
    }
    static void B(int i)
    {
        Console.WriteLine("B called with {0}", i);
    }
    static void Main(string[] args)
    {
        myDelegate md = new myDelegate(A);
        md(1);
        md = new myDelegate(B);
        md(2);
    }
}

The complete program is shown in Figure 9-3. As you might expect, the output when we run this program is identical to that of the previous C++ example.

Figure 9-3

Multicasting

A feature of delegates that is not possible with pointers to functions is multicasting. Multicasting is a feature that allows more than one method to be associated with a delegate instance. The += operator is used to add additional methods. If we replace the four lines of code discussed above with the following three lines we get the output shown in Figure 9-4:

myDelegate md = new myDelegate(A);
md += new myDelegate(B);
md(2);

Notice that one call to md invokes both methods A and B. We can use the -= operator to remove a method from the delegate list. However, this requires that
we use a named instance in order to identify the instance we want. We might do something like this:

```csharp
myDelegate md = new myDelegate(A);
myDelegate del2 = new myDelegate(B);
md += del2;
md(1);
md -= del2; //removes the delegate
md(2);
```

The second call to `md` would output "A called with 2."

![Figure 9-4](image)

**Events**

We have been using the built-in events of the FCL such as for a mouse click or menu item. You can declare your own events and use them in the same way.

An event is really just a delegate with a very specific prototype:

```csharp
Delegate void MyEvent(object sender, MyEventArgs e);
```

This is actually a convention and not mandatory. `sender` refers to the class that is triggering the event. The second argument, `e`, is an object derived from `System.EventArgs` and allows arbitrary information to be passed to the recipient event handler.

Adding the `event` keyword to a delegate instance ensures that only the `+=` and `-=` operators will be used outside the class that declares the event. This prevents the accidental overwriting of handlers that are already assigned to the delegate. Consider using the mouse click event. We can only add or remove a specific handler and not eliminate all existing handlers by the accidental use of the assignment operator. Furthermore, an event can only be triggered from within the class that defines it. We need to provide a method to trigger the event.

The `Control` class defines the following click event:

```csharp
public event EventHandler Click;
```
When we add an event handler for this event manually or using the designer we are adding our handler to any that already exist. Normally this event is triggered by the operating system when you click the mouse. However, you can call a method in the `Control` class to programmatically trigger this event.

```csharp
protected void InvokeOnClick (
    Control toInvoke,
    EventArgs e
)
```

This is a `protected` method and therefore only be used by classes derived from `Control`. Of course any method in your form class can call it since `Form` is derived from `Control`. This is just an example and a trigger method for an event can just as well be `public`.

Our previous example can be rewritten using an event as shown in Figures Figure 9-5 and Figure 9-6.

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;

namespace EventHandler1
{
    public delegate void myEvent(object sender, MyEventArgs e);
    class MyEventArgs : EventArgs
    {
        public int i;
    }

    class Program
    {
        public event myEvent me;
        static void Main(string[] args)
        {
            (new Program()).Run();
        }
        void Run()
        {
            MyEventArgs e = new MyEventArgs();
            e.i = 1;
            me = new myEvent(A);
            Trigger(this, e);
            (new EventHander(this)).addEvent();
        }

        public void Trigger(object sender, MyEventArgs e)
        {
            me(sender, e); //triger the event
        }
        void A(object sender, MyEventArgs e)
        {
```
```
Since we need to pass an argument to the event handlers A and B I have declared my own class derived from EventArgs. It includes a field for the value i. The delegate myEvent is used to declare an event handler. The actual event is declared as myEvent me and is a public field in the Program class. Since I didn’t want to make everything static I instantiate the Program class and call the Run method that does all the work. Run first creates a new MyEventArgs object and sets i to one. Method A is added to the event me. Run then triggers the event by calling the public method Trigger. All Trigger does is to invoke the event me. To demonstrate how another class could add its own event handler to the event and then trigger it, I wrote the tryEvent class. Its constructor is passed a
reference to the *Program* class instance. The *addEvent* method adds the event handle `B` to the `me` event and triggers the event.

What is important to note is that you can only add the event handler using the `+=` operator. If you try to replace the chain of existing event handlers using an assignment the compiler would generate an error. All this is the result of using the *event* keyword. If you were to remove the *event* keyword and uncomment the assignment the compiler would not complain. So you see, events are really just delegates with a recommended signature and a mechanism to safeguard accidentally replacing all the handlers by assignment.

The FCL contains a declaration for a generic event handler that does not require custom event arguments. In that case you can use it without declaring your own custom delegate.

```csharp
public delegate void EventHandler(object sender, EventArgs e);
```