CHAPTER 17

Delegates

In this chapter:

In this chapter, I talk about callback functions. Callback functions are an extremely useful programming mechanism that has been around for years. The Microsoft .NET Framework exposes a callback function mechanism by using *delegates*. Unlike callback mechanisms used in other platforms, such as unmanaged C++, delegates offer much more functionality. For example, delegates ensure that the callback method is type-safe, in keeping with one of the most important goals of the common language runtime (CLR). Delegates also integrate the ability to call multiple methods sequentially and support the calling of static methods as well as instance methods.

A First Look at Delegates

The C runtime's `qsort` function takes a pointer to a callback function to sort elements within an array. In Windows, callback functions are required for window procedures, hook procedures, asynchronous procedure calls, and more. In the .NET Framework, callback methods are used for a whole slew of things. For example, you can register callback methods to get a variety of notifications such as unhandled exceptions, window state changes, menu item selections, file system changes, form control events, and completed asynchronous operations.

In unmanaged C/C++, the address of a non-member function is just a memory address. This address doesn't carry any additional information such as the number of parameters the function expects, the types of these parameters, the function's return value type, and the function's calling convention. In short, unmanaged C/C++ callback functions are not type-safe (although they are a very lightweight mechanism).

In the .NET Framework, callback functions are just as useful and pervasive as in unmanaged Windows programming. However, the .NET Framework provides a type-safe mechanism called *delegates*. I'll start off the discussion of delegates by showing you how to use them. The following code demonstrates how to declare, create, and use delegates.

```
using System;
using System.Windows.Forms;
using System.IO;


// Declare a delegate type; instances refer to a method that
// takes an Int32 parameter and returns void.
internal delegate void Feedback(Int32 value);


public sealed class Program {
   public static void Main() {
      StaticDelegateDemo();
      InstanceDelegateDemo();
      ChainDelegateDemo1(new Program());
      ChainDelegateDemo2(new Program());
   }

   private static void StaticDelegateDemo() {
      Console.WriteLine("----- Static Delegate Demo -----");
      Counter(1, 3, null);
      Counter(1, 3, new Feedback(Program.FeedbackToConsole));
      Counter(1, 3, new Feedback(FeedbackToMsgBox)); // "Program." is optional
      Console.WriteLine();
   }

   private static void InstanceDelegateDemo() {
      Console.WriteLine("----- Instance Delegate Demo -----");
      Program p = new Program();
      Counter(1, 3, new Feedback(p.FeedbackToFile));

      Console.WriteLine();
   }

   private static void ChainDelegateDemo1(Program p) {
      Console.WriteLine("----- Chain Delegate Demo 1 -----");
      Feedback fb1 = new Feedback(FeedbackToConsole);
      Feedback fb2 = new Feedback(FeedbackToMsgBox);
      Feedback fb3 = new Feedback(p.FeedbackToFile);

      Feedback fbChain = null;
      fbChain = (Feedback) Delegate.Combine(fbChain, fb1);
      fbChain = (Feedback) Delegate.Combine(fbChain, fb2);
      fbChain = (Feedback) Delegate.Combine(fbChain, fb3);
      Counter(1, 2, fbChain);

      Console.WriteLine();
      fbChain = (Feedback)
         Delegate.Remove(fbChain, new Feedback(FeedbackToMsgBox));
      Counter(1, 2, fbChain);
   }
```

```
   private static void ChainDelegateDemo2(Program p) {
      Console.WriteLine("----- Chain Delegate Demo 2 -----");
      Feedback fb1 = new Feedback(FeedbackToConsole);
      Feedback fb2 = new Feedback(FeedbackToMsgBox);
      Feedback fb3 = new Feedback(p.FeedbackToFile);

      Feedback fbChain = null;
      fbChain += fb1;
      fbChain += fb2;
      fbChain += fb3;
      Counter(1, 2, fbChain);

      Console.WriteLine();
      fbChain -= new Feedback(FeedbackToMsgBox);
      Counter(1, 2, fbChain);
   }

   private static void Counter(Int32 from, Int32 to, Feedback fb) {
      for (Int32 val = from; val <= to; val++) {
         // If any callbacks are specified, call them
         if (fb != null)
            fb(val);
      }
   }

   private static void FeedbackToConsole(Int32 value) {
      Console.WriteLine("Item=" + value);
   }

   private static void FeedbackToMsgBox(Int32 value) {
      MessageBox.Show("Item=" + value);
   }

   private void FeedbackToFile(Int32 value) {
      using (StreamWriter sw = new StreamWriter("Status", true)) {
         sw.WriteLine("Item=" + value);
      }
   }
}
```

Now I'll describe what this code is doing. At the top, notice the declaration of the internal delegate, `Feedback`. A delegate indicates the signature of a callback method. In this example, a `Feedback` delegate identifies a method that takes one parameter (an `Int32`) and returns `void`. In a way, a delegate is very much like an unmanaged C/C++ `typedef` that represents the address of a function.

The `Program` class defines a private, static method named `Counter`. This method counts integers from the `from` argument to the `to` argument. The `Counter` method also takes an `fb`, which is a reference to a `Feedback` delegate object. `Counter` iterates through all of the integers, and for each integer, if the `fb` variable is not `null`, the callback method (specified by the `fb` variable) is called. This callback method is passed the value of the item being processed, the item number. The callback method can be designed and implemented to process each item in any manner deemed appropriate.

Using Delegates to Call Back Static Methods

Now that you understand how the `Counter` method is designed and how it works, let's see how to use delegates to call back static methods. The `StaticDelegateDemo` method that appears in the previous code sample is the focus of this section.

The `StaticDelegateDemo` method calls the `Counter` method, passing `null` in the third parameter, which corresponds to `Counter`'s `fb` parameter. Because `Counter`'s `fb` parameter receives `null`, each item is processed without calling any callback method.

Next, the `StaticDelegateDemo` method calls `Counter` a second time, passing a newly constructed `Feedback` delegate object in the third parameter of the method call. This delegate object is a wrapper around a method, allowing the method to be called back indirectly via the wrapper. In this example, the name of the static method, `Program.FeedbackToConsole`, is passed to the `Feedback` type's constructor, indicating that it is the method to be wrapped. The reference returned from the `new` operator is passed to `Counter` as its third parameter. Now when `Counter` executes, it will call the `Program` type's static `FeedbackToConsole` method for each item in the series. `FeedbackToConsole` simply writes a string to the console indicating the item being processed.

Note The `FeedbackToConsole` method is defined as `private` inside the `Program` type, but the `Counter` method is able to call `Program`'s private method. In this case, you might not expect a problem because both `Counter` and `FeedbackToConsole` are defined in the same type. However, this code would work just fine even if the `Counter` method was defined in another type. In short, it is not a security or accessibility violation for one type to have code that calls another type's private member via a delegate as long as the delegate object is created by code that has ample security/accessibility.

The third call to `Counter` in the `StaticDelegateDemo` method is almost identical to the second call. The only difference is that the `Feedback` delegate object wraps the static `Program.FeedbackToMsgBox` method. `FeedbackToMsgBox` builds a string indicating the item being processed. This string is then displayed in a message box.

Everything in this example is type-safe. For instance, when constructing a `Feedback` delegate object, the compiler ensures that the signatures of `Program`'s `FeedbackToConsole` and `FeedbackToMsgBox` methods are compatible with the signature defined by the `Feedback` delegate. Specifically, both methods must take one argument (an `Int32`), and both methods must have the same return type (`void`). If `FeedbackToConsole` had been defined like this:

```
private static Boolean FeedbackToConsole(String value) {
   ...
}
```

the C# compiler wouldn't compile the code and would issue the following error: `error CS0123: No overload for 'FeedbackToConsole' matches delegate 'Feedback'.`

Both C# and the CLR allow for covariance and contra-variance of reference types when binding a method to a delegate. Covariance means that a method can return a type that is derived from the delegate's return type. Contra-variance means that a method can take a parameter that is a base of the delegate's parameter type. For example, given a delegate defined like this:

```
delegate Object MyCallback(FileStream s);
```

it is possible to construct an instance of this delegate type bound to a method that is prototyped like this.

```
String SomeMethod(Stream s);
```

Here, `SomeMethod`'s return type (`String`) is a type that is derived from the delegate's return type (`Object`); this covariance is allowed. `SomeMethod`'s parameter type (`Stream`) is a type that is a base class of the delegate's parameter type (`FileStream`); this contra-variance is allowed.

Note that covariance and contra-variance are supported only for reference types, not for value types or for `void`. So, for example, I cannot bind the following method to the `MyCallback` delegate.

```
Int32 SomeOtherMethod(Stream s);
```

Even though `SomeOtherMethod`'s return type (`Int32`) is derived from `MyCallback`'s return type (`Object`), this form of covariance is not allowed because `Int32` is a value type. Obviously, the reason why value types and `void` cannot be used for covariance and contra-variance is because the memory structure for these things varies, whereas the memory structure for reference types is always a pointer. Fortunately, the C# compiler will produce an error if you attempt to do something that is not supported.

Using Delegates to Call Back Instance Methods

I just explained how delegates can be used to call static methods, but they can also be used to call instance methods for a specific object. To understand how calling back an instance method works, look at the `InstanceDelegateDemo` method that appears in the code shown at the beginning of this chapter.

Notice that a `Program` object named `p` is constructed in the `InstanceDelegateDemo` method. This `Program` object doesn't have any instance fields or properties associated with it; I created it merely for demonstration purposes. When the new `Feedback` delegate object is constructed in the call to the `Counter` method, its constructor is passed `p.FeedbackToFile`. This causes the delegate to wrap a reference to the `FeedbackToFile` method, which is an instance method (not a static method). When `Counter` calls the callback method identified by its `fb` argument, the `FeedbackToFile` instance method is called, and the address of the recently constructed object `p` will be passed as the implicit `this` argument to the instance method.

The FeedbackToFile method works as the FeedbackToConsole and FeedbackToMsgBox methods, except that it opens a file and appends the string to the end of the file. (The Status file that the method creates can be found in the application's AppBase directory.)

Again, the purpose of this example is to demonstrate that delegates can wrap calls to instance methods as well as static methods. For instance methods, the delegate needs to know the instance of the object the method is going to operate on. Wrapping an instance method is useful because code inside the object can access the object's instance members. This means that the object can have some state that can be used while the callback method is doing its processing.

Demystifying Delegates

On the surface, delegates seem easy to use: you define them by using C#'s `delegate` keyword, you construct instances of them by using the familiar `new` operator, and you invoke the callback by using the familiar method-call syntax (except instead of a method name, you use the variable that refers to the delegate object).

However, what's really going on is quite a bit more complex than what the earlier examples illustrate. The compilers and the CLR do a lot of behind-the-scenes processing to hide the complexity. In this section, I'll focus on how the compiler and the CLR work together to implement delegates. Having this knowledge will improve your understanding of delegates and will teach you how to use them efficiently and effectively. I'll also touch on some additional features delegates make available.

Let's start by reexamining this line of code.

```
internal delegate void Feedback(Int32 value);
```

When the compiler sees this line, it actually defines a complete class that looks something like this.

```
internal class Feedback : System.MulticastDelegate {
   // Constructor
   public Feedback(Object @object, IntPtr method);

   // Method with same prototype as specified by the source code
   public virtual void Invoke(Int32 value);

   // Methods allowing the callback to be called asynchronously
   public virtual IAsyncResult BeginInvoke(Int32 value,
      AsyncCallback callback, Object @object);
   public virtual void EndInvoke(IAsyncResult result);
}
```

The class defined by the compiler has four methods: a constructor, `Invoke`, `BeginInvoke`, and `EndInvoke`. In this chapter, I'll concentrate on the constructor and `Invoke` methods. The `BeginInvoke` and `EndInvoke` methods are related to the .NET Framework's Asynchronous Programming Model which is now considered obsolete and has been replaced by tasks that I discuss in Chapter 27, "Compute-Bound Asynchronous Operations."

In fact, you can verify that the compiler did indeed generate this class automatically by examining the resulting assembly with ILDasm.exe, as shown in Figure 17-1.

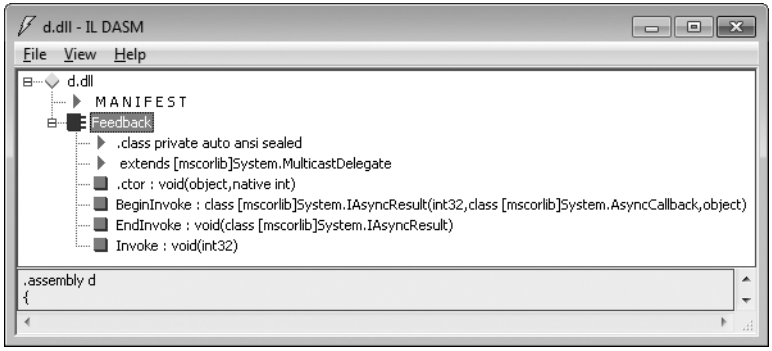

FIGURE 17-1 ILDasm.exe showing the metadata produced by the compiler for the delegate.

In this example, the compiler has defined a class called `Feedback` that is derived from the `System.MulticastDelegate` type defined in the Framework Class Library (FCL). (All delegate types are derived from `MulticastDelegate`.)

Important The `System.MulticastDelegate` class is derived from `System.Delegate`, which is itself derived from `System.Object`. The reason why there are two delegate classes is historical and unfortunate; there should be just one delegate class in the FCL. Sadly, you need to be aware of both of these classes because even though all delegate types you create have `MulticastDelegate` as a base class, you'll occasionally manipulate your delegate types by using methods defined by the `Delegate` class instead of the `MulticastDelegate` class. For example, the `Delegate` class has static methods called `Combine` and `Remove`. (I explain what these methods do later.) The signatures for both of these methods indicate that they take `Delegate` parameters. Because your delegate type is derived from `MulticastDelegate`, which is derived from `Delegate`, instances of your delegate type can be passed to these methods.

The class has private visibility because the delegate is declared as `internal` in the source code. If the source code had indicated `public` visibility, the `Feedback` class the compiler generated would also be public. You should be aware that delegate types can be defined within a type (nested within another type) or at global scope. Basically, because delegates are classes, a delegate can be defined anywhere a class can be defined.

Because all delegate types are derived from `MulticastDelegate`, they inherit `MulticastDelegate`'s fields, properties, and methods. Of all of these members, three non-public fields are probably most significant. Table 17-1 describes these significant fields.

TABLE 17-1 `MulticastDelegate`'s Significant Non-Public Fields

Field	Type	Description
`_target`	`System.Object`	When the delegate object wraps a static method, this field is `null`. When the delegate objects wraps an instance method, this field refers to the object that should be operated on when the callback method is called. In other words, this field indicates the value that should be passed for the instance method's implicit `this` parameter.
`_methodPtr`	`System.IntPtr`	An internal integer the CLR uses to identify the method that is to be called back.
`_invocationList`	`System.Object`	This field is usually null. It can refer to an array of delegates when building a delegate chain (discussed later in this chapter).

Notice that all delegates have a constructor that takes two parameters: a reference to an object and an integer that refers to the callback method. However, if you examine the source code, you'll see that I'm passing in values such as `Program.FeedbackToConsole` or `p.FeedbackToFile`. Everything you've learned about programming tells you that this code shouldn't compile!

However, the C# compiler knows that a delegate is being constructed and parses the source code to determine which object and method are being referred to. A reference to the object is passed for the constructor's `object` parameter, and a special `IntPtr` value (obtained from a `MethodDef` or `MemberRef` metadata token) that identifies the method is passed for the `method` parameter. For static methods, `null` is passed for the `object` parameter. Inside the constructor, these two arguments are saved in the `_target` and `_methodPtr` private fields, respectively. In addition, the constructor sets the `_invocationList` field to `null`. I'll postpone discussing this `_invocationList` field until the next section, "Using Delegates to Call Back Many Methods (Chaining)."

So each delegate object is really a wrapper around a method and an object to be operated on when the method is called. So if I have two lines of code that look like this:

```
Feedback fbStatic   = new Feedback(Program.FeedbackToConsole);
Feedback fbInstance = new Feedback(new Program().FeedbackToFile);
```

the `fbStatic` and `fbInstance` variables refer to two separate `Feedback` delegate objects that are initialized, as shown in Figure 17-2.

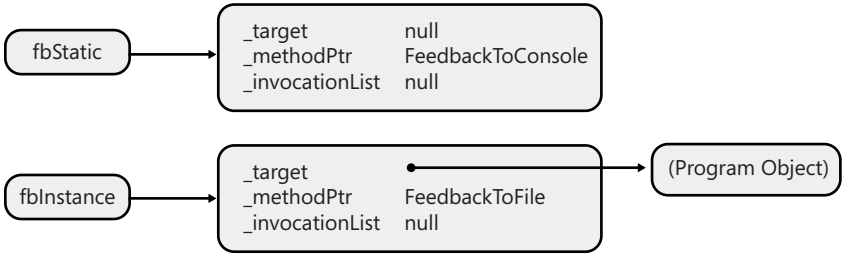

FIGURE 17-2 A variable that refers to a delegate to a static method and a variable that refers to a delegate to an instance method.

Now that you know how delegate objects are constructed and what their internal structure looks like, let's talk about how the callback method is invoked. For convenience, I've repeated the code for the `Counter` method here.

```
private static void Counter(Int32 from, Int32 to, Feedback fb) {
   for (Int32 val = from; val <= to; val++) {
      // If any callbacks are specified, call them
      if (fb != null)
         fb(val);
   }
}
```

Look at the line of code just below the comment. The `if` statement first checks to see if `fb` is not `null`. If `fb` is not `null`, on the next line, you see the code that invokes the callback method. The `null` check is required because `fb` is really just a variable that can refer to a `Feedback` delegate object; it could also be `null`. It might seem as if I'm calling a function named `fb` and passing it one parameter (`val`). However, there is no function called `fb`. Again, because it knows that `fb` is a variable that refers to a delegate object, the compiler generates code to call the delegate object's `Invoke` method. In other words, the compiler sees the following.

```
fb(val);
```

But the compiler generates code as though the source code said the following.

```
fb.Invoke(val);
```

You can verify that the compiler produces code to call the delegate type's `Invoke` method by using ILDasm.exe to examine the Intermediate Language (IL) code created for the `Counter` method. Here is the IL for the `Counter` method. The instruction at IL_0009 in the figure indicates the call to `Feedback`'s `Invoke` method.

```
.method private hidebysig static void  Counter(int32 from,
                                               int32 'to',
                                               class Feedback fb) cil managed
{
  // Code size       23 (0x17)
  .maxstack  2
  .locals init (int32 val)
  IL_0000:  ldarg.0
  IL_0001:  stloc.0
  IL_0002:  br.s       IL_0012
  IL_0004:  ldarg.2
  IL_0005:  brfalse.s  IL_000e
  IL_0007:  ldarg.2
  IL_0008:  ldloc.0
  IL_0009:  callvirt   instance void Feedback::Invoke(int32)
  IL_000e:  ldloc.0
  IL_000f:  ldc.i4.1
  IL_0010:  add
  IL_0011:  stloc.0
  IL_0012:  ldloc.0
  IL_0013:  ldarg.1
  IL_0014:  ble.s      IL_0004
  IL_0016:  ret
} // end of method Program::Counter
```

In fact, you could modify the `Counter` method to call `Invoke` explicitly, as shown here.

```
private static void Counter(Int32 from, Int32 to, Feedback fb) {
   for (Int32 val = from; val <= to; val++) {
      // If any callbacks are specified, call them
      if (fb != null)
         fb.Invoke(val);
   }
}
```

You'll recall that the compiler defined the `Invoke` method when it defined the `Feedback` class. When `Invoke` is called, it uses the private `_target` and `_methodPtr` fields to call the desired method on the specified object. Note that the signature of the `Invoke` method matches the signature of the delegate; because the `Feedback` delegate takes one `Int32` parameter and returns `void`, the `Invoke` method (as produced by the compiler) takes one `Int32` parameter and returns `void`.

Using Delegates to Call Back Many Methods (Chaining)

By themselves, delegates are incredibly useful. But add in their support for chaining, and delegates become even more useful. Chaining is a set or collection of delegate objects, and it provides the ability to invoke, or call, all of the methods represented by the delegates in the set. To understand this, see the `ChainDelegateDemo1` method that appears in the code shown at the beginning of this chapter. In this method, after the `Console.WriteLine` statement, I construct three delegate objects and have variables—`fb1`, `fb2`, and `fb3`—refer to each object, as shown in Figure 17-3.

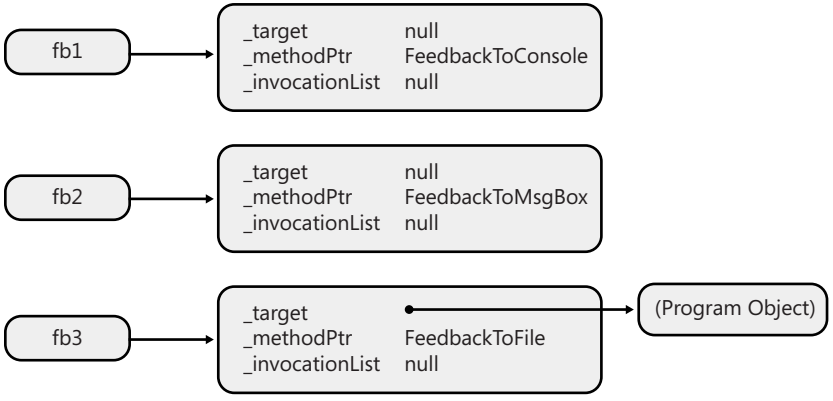

FIGURE 17-3 Initial state of the delegate objects referred to by the `fb1`, `fb2`, and `fb3` variables.

The reference variable to a `Feedback` delegate object, `fbChain`, is intended to refer to a chain or set of delegate objects that wrap methods that can be called back. Initializing `fbChain` to `null` indicates that there currently are no methods to be called back. The `Delegate` class's public, static `Combine` method is used to add a delegate to the chain.

```
fbChain = (Feedback) Delegate.Combine(fbChain, fb1);
```

When this line of code executes, the Combine method sees that we are trying to combine null and fb1. Internally, Combine will simply return the value in fb1, and the fbChain variable will be set to refer to the same delegate object referred to by the fb1 variable, as shown in Figure 17-4.

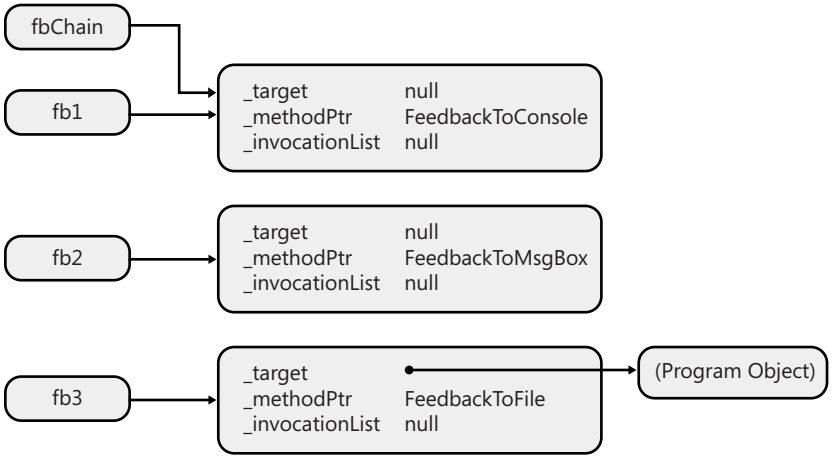

FIGURE 17-4 State of the delegate objects after inserting the first delegate in the chain.

To add another delegate to the chain, the Combine method is called again.

```
fbChain = (Feedback) Delegate.Combine(fbChain, fb2);
```

Internally, the Combine method sees that fbChain already refers to a delegate object, so Combine will construct a new delegate object. This new delegate object initializes its private _target and _methodPtr fields to values that are not important for this discussion. However, what is important is that the _invocationList field is initialized to refer to an array of delegate objects. The first element of this array (index 0) will be initialized to refer to the delegate that wraps the FeedbackToConsole method (this is the delegate that fbChain currently refers to). The second element of the array (index 1) will be initialized to refer to the delegate that wraps the FeedbackToMsgBox method (this is the delegate that fb2 refers to). Finally, fbChain will be set to refer to the newly created delegate object, shown in Figure 17-5.

To add the third delegate to the chain, the Combine method is called once again.

```
fbChain = (Feedback) Delegate.Combine(fbChain, fb3);
```

Again, Combine sees that fbChain already refers to a delegate object, and this causes a new delegate object to be constructed, as shown in Figure 17-6. As before, this new delegate object initializes the private _target and _methodPtr fields to values unimportant to this discussion, and the _invocationList field is initialized to refer to an array of delegate objects. The first and second elements of this array (indexes 0 and 1) will be initialized to refer to the same delegates the previous delegate object referred to in its array. The third element of the array (index 2) will be initialized to refer to the delegate that wraps the FeedbackToFile method (this is the delegate that fb3 refers to). Finally, fbChain will be set to refer to this newly created delegate object. Note that the previously created delegate and the array referred to by its _invocationList field are now candidates for garbage collection.

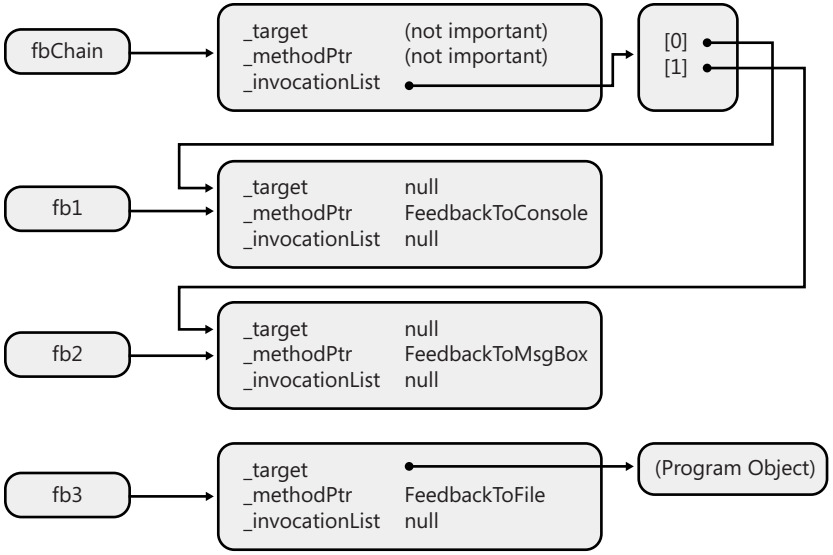

FIGURE 17-5 State of the delegate objects after inserting the second delegate in the chain.

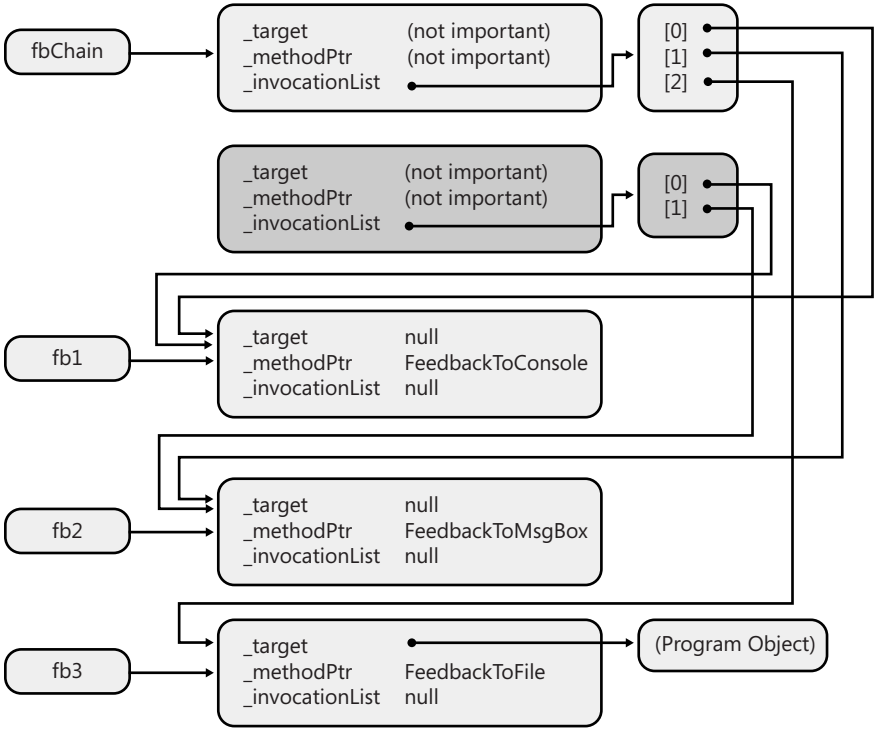

FIGURE 17-6 Final state of the delegate objects when the chain is complete.

After all of the code has executed to set up the chain, the fbChain variable is then passed to the Counter method.

```
Counter(1, 2, fbChain);
```

Inside the Counter method is the code that implicitly calls the Invoke method on the Feedback delegate object as I detailed earlier. When Invoke is called on the delegate referred to by fbChain, the delegate sees that the private _invocationList field is not null, causing it to execute a loop that iterates through all of the elements in the array, calling the method wrapped by each delegate. In this example, FeedbackToConsole will get called first, followed by FeedbackToMsgBox, followed by FeedbackToFile.

Feedback's Invoke method is essentially implemented something like the following (in pseudo-code).

```
public void Invoke(Int32 value) {
   Delegate[] delegateSet = _invocationList as Delegate[];
   if (delegateSet != null) {
      // This delegate's array indicates the delegates that should be called
      foreach (Feedback d in delegateSet)
         d(value);   // Call each delegate
   } else {
      // This delegate identifies a single method to be called back
      // Call the callback method on the specified target object.
      _methodPtr.Invoke(_target, value);
      // The preceding line is an approximation of the actual code.
      // What really happens cannot be expressed in C#.
   }
}
```

Note that it is also possible to remove a delegate from a chain by calling Delegate's public, static Remove method. This is demonstrated toward the end of the ChainDelegateDemo1 method.

```
fbChain = (Feedback) Delegate.Remove(fbChain, new Feedback(FeedbackToMsgBox));
```

When Remove is called, it scans the delegate array (from the end toward index 0) maintained inside the delegate object referred to by the first parameter (fbChain, in my example). Remove is looking for a delegate entry whose _target and _methodPtr fields match those in the second argument (the new Feedback delegate, in my example). If a match is found and there is only one item left in the array, that array item is returned. If a match is found and there are multiple items left in the array, a new delegate object is constructed—the _invocationList array created and initialized will refer to all items in the original array except for the item being removed, of course—and a reference to this new delegate object is returned. If you are removing the only element in the chain, Remove returns null. Note that each call to Remove removes just one delegate from the chain; it does not remove all delegates that have matching _target and _methodPtr fields.

So far, I've shown examples in which my delegate type, Feedback, is defined as having a void return value. However, I could have defined my Feedback delegate as follows.

```
public delegate Int32 Feedback(Int32 value);
```

If I had, its `Invoke` method would have internally looked like the following (again, in pseudocode).

```
public Int32 Invoke(Int32 value) {
   Int32 result;
   Delegate[] delegateSet = _invocationList as Delegate[];
   if (delegateSet != null) {
      // This delegate's array indicates the delegates that should be called
      foreach (Feedback d in delegateSet)
         result = d(value);   // Call each delegate
   } else {
      // This delegate identifies a single method to be called back
      // Call the callback method on the specified target object.
      result = _methodPtr.Invoke(_target, value);
      // The preceding line is an approximation of the actual code.
      // What really happens cannot be expressed in C#.
   }
   return result;
}
```

As each delegate in the array is called, its return value is saved in the result variable. When the loop is complete, the `result` variable will contain only the result of the last delegate called (previous return values are discarded); this value is returned to the code that called `Invoke`.

C#'s Support for Delegate Chains

To make things easier for C# developers, the C# compiler automatically provides overloads of the += and -= operators for instances of delegate types. These operators call `Delegate.Combine` and `Delegate.Remove`, respectively. Using these operators simplifies the building of delegate chains. The `ChainDelegateDemo1` and `ChainDelegateDemo2` methods in the source code shown at the beginning of this chapter produce absolutely identical IL code. The only difference between the methods is that the `ChainDelegateDemo2` method simplifies the source code by taking advantage of C#'s += and -= operators.

If you require proof that the resulting IL code is identical for the two methods, you can build the code and look at its IL for both methods by using ILDasm.exe. This will confirm that the C# compiler did in fact replace all += and -= operators with calls to the `Delegate` type's public static `Combine` and `Remove` methods, respectively.

Having More Control over Delegate Chain Invocation

At this point, you understand how to build a chain of delegate objects and how to invoke all of the objects in that chain. All items in the chain are invoked because the delegate type's `Invoke` method includes code to iterate through all of the items in the array, invoking each item. This is obviously a very simple algorithm. And although this simple algorithm is good enough for a lot of scenarios, it has many limitations. For example, the return values of the callback methods are all discarded except for the last one. Using this simple algorithm, there's no way to get the return values for all of the callback methods called. But this isn't the only limitation. What happens if one of the invoked delegates throws an exception or blocks for a very long time? Because the algorithm invoked each delegate in

the chain serially, a "problem" with one of the delegate objects stops all of the subsequent delegates in the chain from being called. Clearly, this algorithm isn't robust.

For those scenarios in which this algorithm is insufficient, the `MulticastDelegate` class offers an instance method, `GetInvocationList`, that you can use to call each delegate in a chain explicitly, using any algorithm that meets your needs.

```
public abstract class MulticastDelegate : Delegate {
   // Creates a delegate array where each element refers
   // to a delegate in the chain.
   public sealed override Delegate[] GetInvocationList();
}
```

The `GetInvocationList` method operates on a `MulticastDelegate`-derived object and returns an array of `Delegate` references where each reference points to one of the chain's delegate objects. Internally, `GetInvocationList` constructs an array and initializes it with each element referring to a delegate in the chain; a reference to the array is then returned. If the `_invocationList` field is `null`, the returned array contains one element that references the only delegate in the chain: the delegate instance itself.

You can easily write an algorithm that explicitly calls each object in the array. The following code demonstrates.

```
using System;
using System.Reflection;
using System.Text;

// Define a Light component.
internal sealed class Light {
   // This method returns the light's status.
   public String SwitchPosition() {
      return "The light is off";
   }
}

// Define a Fan component.
internal sealed class Fan {
   // This method returns the fan's status.
   public String Speed() {
      throw new InvalidOperationException("The fan broke due to overheating");
   }
}

// Define a Speaker component.
internal sealed class Speaker {
   // This method returns the speaker's status.
   public String Volume() {
      return "The volume is loud";
   }
}

public sealed class Program {
```

```
   // Definition of delegate that allows querying a component's status.
   private delegate String GetStatus();

   public static void Main() {
      // Declare an empty delegate chain.
      GetStatus getStatus = null;

      // Construct the three components, and add their status methods
      // to the delegate chain.
      getStatus += new GetStatus(new Light().SwitchPosition);
      getStatus += new GetStatus(new Fan().Speed);
      getStatus += new GetStatus(new Speaker().Volume);

      // Show consolidated status report reflecting
      // the condition of the three components.
      Console.WriteLine(GetComponentStatusReport(getStatus));
   }

   // Method that queries several components and returns a status report
   private static String GetComponentStatusReport(GetStatus status) {

      // If the chain is empty, there is nothing to do.
      if (status == null) return null;

      // Use this to build the status report.
      StringBuilder report = new StringBuilder();

      // Get an array where each element is a delegate from the chain.
      Delegate[] arrayOfDelegates = status.GetInvocationList();

      // Iterate over each delegate in the array.
      foreach (GetStatus getStatus in arrayOfDelegates) {

         try {
            // Get a component's status string, and append it to the report.
            report.AppendFormat("{0}{1}{1}", getStatus(), Environment.NewLine);
         }
         catch (InvalidOperationException e) {
            // Generate an error entry in the report for this component.
            Object component = getStatus.Target;
            report.AppendFormat(
               "Failed to get status from {1}{2}{0}   Error: {3}{0}{0}",
               Environment.NewLine,
               ((component == null) ? "" : component.GetType() + "."),
               getStatus.GetMethodInfo().Name,
               e.Message);
         }
      }

      // Return the consolidated report to the caller.
      return report.ToString();
   }
}
```

When you build and run this code, the following output appears.

```
The light is off

Failed to get status from Fan.Speed
   Error: The fan broke due to overheating

The volume is loud
```

Enough with the Delegate Definitions Already (Generic Delegates)

Many years ago, when the .NET Framework was just starting to be developed, Microsoft introduced the notion of delegates. As programmers were adding classes to the FCL, they would define new delegate types any place they introduced a callback method. Over time, many, many delegates got defined. In fact, in MSCorLib.dll alone, close to 50 delegate types are now defined. Let's just look at a few of them.

```
public delegate void TryCode(Object userData);
public delegate void WaitCallback(Object state);
public delegate void TimerCallback(Object state);
public delegate void ContextCallback(Object state);
public delegate void SendOrPostCallback(Object state);
public delegate void ParameterizedThreadStart(Object obj);
```

Do you notice anything similar about the few delegate definitions that I selected? They are really all the same: a variable of any of these delegate types must refer to a method that takes an `Object` and returns `void`. There is really no reason to have all of these delegate types defined; there really just needs to be one.

In fact, now that the .NET Framework supports generics, we really just need a few generic delegates (defined in the `System` namespace) that represent methods that take up to 16 arguments.

```
public delegate void Action();       // OK, this one is not generic
public delegate void Action<T>(T obj);
public delegate void Action<T1, T2>(T1 arg1, T2 arg2);
public delegate void Action<T1, T2, T3>(T1 arg1, T2 arg2, T3 arg3);
...
public delegate void Action<T1, ..., T16>(T1 arg1, ..., T16 arg16);
```

So the .NET Framework now ships with 17 `Action` delegates that range from having no arguments to having 16 arguments. If you ever need to call a method that has more than 16 arguments, you will be forced to define your own delegate type, but this is very unlikely.

In addition to the `Action` delegates, the .NET Framework ships with 17 `Func` delegates, which allow the callback method to return a value.

```
public delegate TResult Func<TResult>();
public delegate TResult Func<T, TResult>(T arg);
public delegate TResult Func<T1, T2, TResult>(T1 arg1, T2 arg2);
public delegate TResult Func<T1, T2, T3, TResult>(T1 arg1, T2 arg2, T3 arg3);
...
public delegate TResult Func<T1,..., T16, TResult>(T1 arg1, ..., T16 arg16);
```

It is now recommended that these delegate types be used wherever possible instead of developers defining even more delegate types in their code. This reduces the number of types in the system and also simplifies coding. However, you might have to define your own delegate if you need to pass an argument by reference using the `ref` or `out` keyword.

```
delegate void Bar(ref Int32 z);
```

You may also have to do this if you want your delegate to take a variable number of arguments via C#'s `params` keyword, if you want to specify any default values for any of your delegate's arguments, or if you need to constrain a delegate's generic type argument.

When using delegates that take generic arguments and return values, contra-variance and co-variance come into play, and it is recommended that you always take advantage of these features because they have no ill effects and enable your delegates to be used in more scenarios. For more information about this, see the "Delegate and Interface Contra-variant and Covariant Generic Type Arguments" section in Chapter 12, "Generics."

C#'s Syntactical Sugar for Delegates

Most programmers find working with delegates to be cumbersome because the syntax is so strange. For example, take this line of code.

```
button1.Click += new EventHandler(button1_Click);
```

where `button1_Click` is a method that looks something like this.

```
void button1_Click(Object sender, EventArgs e) {
   // Do something, the button was clicked...
}
```

The idea behind the first line of code is to register the address of the `button1_Click` method with a button control so that when the button is clicked, the method will be called. To most programmers, it feels quite unnatural to construct an `EventHandler` delegate object just to specify the

address of the button1_Click method. However, constructing the EventHandler delegate object is required for the CLR because this object provides a wrapper that ensures that the method can be called only in a type-safe fashion. The wrapper also allows the calling of instance methods and chaining. Unfortunately, most programmers don't want to think about these details. Programmers would prefer to write the preceding code as follows.

```
button1.Click += button1_Click;
```

Fortunately, Microsoft's C# compiler offers programmers some syntax shortcuts when working with delegates. I'll explain all of these shortcuts in this section. One last point before we begin: what I'm about to describe really boils down to C# syntactical sugar; these new syntax shortcuts are really just giving programmers an easier way to produce the IL that must be generated so that the CLR and other programming languages can work with delegates. This also means that what I'm about to describe is specific to C#; other compilers might not offer the additional delegate syntax shortcuts.

Syntactical Shortcut #1: No Need to Construct a Delegate Object

As demonstrated already, C# allows you to specify the name of a callback method without having to construct a delegate object wrapper. Here is another example.

```
internal sealed class AClass {
   public static void CallbackWithoutNewingADelegateObject() {
      ThreadPool.QueueUserWorkItem(SomeAsyncTask, 5);
   }

   private static void SomeAsyncTask(Object o) {
      Console.WriteLine(o);
   }
}
```

Here, the ThreadPool class's static QueueUserWorkItem method expects a reference to a WaitCallback delegate object that contains a reference to the SomeAsyncTask method. Because the C# compiler is capable of inferring this on its own, it allows me to omit code that constructs the WaitCallback delegate object, making the code much more readable and understandable. Of course, when the code is compiled, the C# compiler does produce IL that does, in fact, new up the WaitCallback delegate object—we just got a syntactical shortcut.

Syntactical Shortcut #2: No Need to Define a Callback Method (Lambda Expressions)

In the preceding code, the name of the callback method, `SomeAsyncTask`, is passed to the `ThreadPool`'s `QueueUserWorkItem` method. C# allows you to write the code for the callback method inline so it doesn't have to be written inside its very own method. For example, the preceding code could be rewritten as follows.

```
internal sealed class AClass {
   public static void CallbackWithoutNewingADelegateObject() {
      ThreadPool.QueueUserWorkItem( obj => Console.WriteLine(obj), 5);
   }
}
```

Notice that the first "argument" to the `QueueUserWorkItem` method is code (which I italicized)! More formally, the italicized code is called a C# *lambda expression*, and it is easy to detect due to the use of C#'s lambda expression operator: =>. You may use a lambda expression in your code where the compiler would normally expect to see a delegate. And, when the compiler sees the use of this lambda expression, the compiler automatically defines a new private method in the class (`AClass`, in this example). This new method is called an *anonymous function* because the compiler creates the name of the method for you automatically, and normally, you wouldn't know its name. However, you could use a tool such as ILDasm.exe to examine the compiler-generated code. After I wrote the preceding code and compiled it, I was able to see, by using ILDasm.exe, that the C# compiler decided to name this method `<CallbackWithoutNewingADelegateObject>b__0` and ensured that this method took a single `Object` argument and returned `void`.

The compiler chose to start the method name with a < sign because in C#, an identifier cannot contain a < sign; this ensures that you will not accidentally define a method that coincides with the name the compiler has chosen for you. Incidentally, while C# forbids identifiers to contain a < sign, the CLR allows it, and that is why this works. Also, note that although you could access the method via reflection by passing the method name as a string, the C# language specification states that there is no guarantee of how the compiler generates the name. For example, each time you compile the code, the compiler could produce a different name for the method.

Using ILDasm.exe, you might also notice that the C# compiler applies the `System.Runtime.CompilerServices.CompilerGeneratedAttribute` attribute to this method to indicate to various tools and utilities that this method was produced by a compiler as opposed to a programmer. The code to the right of the => operator is then placed in this compiler-generated method.

Note When writing a lambda expression, there is no way to apply your own custom attribute to the compiler-generated method. Furthermore, you cannot apply any method modifiers (such as `unsafe`) to the method. But this is usually not a problem because anonymous methods generated by the compiler always end up being private, and the method is either static or nonstatic depending on whether the method accesses any instance members. So there is no need to apply modifiers such as `public`, `protected`, `internal`, `virtual`, `sealed`, `override`, or `abstract` to the method.

Finally, if you write the preceding code and compile it, it's as if the C# compiler rewrote your code to look like the following (comments inserted by me).

```
internal sealed class AClass {
   // This private field is created to cache the delegate object.
   // Pro: CallbackWithoutNewingADelegateObject will not create
   //      a new object each time it is called.
   // Con: The cached object never gets garbage collected
   [CompilerGenerated]
   private static WaitCallback <>9__CachedAnonymousMethodDelegate1;

   public static void CallbackWithoutNewingADelegateObject() {
      if (<>9__CachedAnonymousMethodDelegate1 == null) {
         // First time called, create the delegate object and cache it.
         <>9__CachedAnonymousMethodDelegate1 =
            new WaitCallback(<CallbackWithoutNewingADelegateObject>b__0);
      }
      ThreadPool.QueueUserWorkItem(<>9__CachedAnonymousMethodDelegate1, 5);
   }

   [CompilerGenerated]
   private static void <CallbackWithoutNewingADelegateObject>b__0(Object obj) {
      Console.WriteLine(obj);
   }
}
```

The lambda expression must match that of the `WaitCallback` delegate: it returns `void` and takes an `Object` parameter. However, I specified the name of the parameter by simply putting `obj` to the left of the => operator. On the right of the => operator, `Console.WriteLine` happens to return `void`. However, if I had placed an expression that did not return `void`, the compiler-generated code would just ignore the return value because the method that the compiler generates must have a `void` return type to satisfy the `WaitCallback` delegate.

It is also worth noting that the anonymous function is marked as `private`; this forbids any code not defined within the type from accessing the method (although reflection will reveal that the method does exist). Also, note that the anonymous method is marked as `static`; this is because the code doesn't access any instance members (which it can't because `CallbackWithoutNewingADelegate-Object` is itself a static method. However, the code can reference any static fields or static methods defined within the class. Here is an example.

```
internal sealed class AClass {
   private static String sm_name;  // A static field

   public static void CallbackWithoutNewingADelegateObject() {
      ThreadPool.QueueUserWorkItem(
         // The callback code can reference static members.
         obj =>Console.WriteLine(sm_name + ": " + obj),
         5);
   }
}
```

If the `CallbackWithoutNewingADelegateObject` method had not been static, the anonymous method's code could contain references to instance members. If it doesn't contain references to instance members, the compiler will still produce a static anonymous method because this is more efficient than an instance method because the additional `this` parameter is not necessary. But, if the anonymous method's code does reference an instance member, the compiler will produce a nonstatic anonymous method.

```
internal sealed class AClass {
   private String m_name;  // An instance field

   // An instance method
   public void CallbackWithoutNewingADelegateObject() {
      ThreadPool.QueueUserWorkItem(
         // The callback code can reference instance members.
         obj => Console.WriteLine(m_name + ": " + obj),
         5);
   }
}
```

On the left side of the => operator is where you specify the names of any arguments that are to be passed to the lambda expression. There are some rules you must follow here. See the following examples.

```
// If the delegate takes no arguments, use ()
Func<String> f = () => "Jeff";

// If the delegate takes 1+ arguments, you can explicitly specify the types
Func<Int32, String> f2 = (Int32 n) => n.ToString();
Func<Int32, Int32, String> f3 = (Int32 n1, Int32 n2) => (n1 + n2).ToString();

// If the delegate takes 1+ arguments, the compiler can infer the types
Func<Int32, String> f4 = (n) => n.ToString();
Func<Int32, Int32, String> f5 = (n1, n2) => (n1 + n2).ToString();

// If the delegate takes 1 argument, you can omit the ()s
Func<Int32, String> f6 = n => n.ToString();

// If the delegate has ref/out arguments, you must explicitly specify ref/out and the type
Bar b = (out Int32 n) => n = 5;
```

For the last example, assume that `Bar` is defined as follows.

```
delegate void Bar(out Int32 z);
```

On the right side of the => operator is where you specify the anonymous function body. It is very common for the body to consist of a simple or complex expression that ultimately returns a non-`void` value. In the preceding code, I was assigning lambda expressions that returned `String`s to all the `Func` delegate variables. It is also quite common for the body to consist of a single statement. An example of this is when I called `ThreadPool.QueueUserWorkItem`, passing it a lambda expression that called `Console.WriteLine` (which returns `void`).

If you want the body to consist of two or more statements, then you must enclose it in curly braces. And if the delegate expects a return value, then you must have a `return` statement inside the body. Here is an example.

```
Func<Int32, Int32, String> f7 = (n1, n2) => { Int32 sum = n1 + n2; return sum.ToString(); };
```

Important In case it's not obvious, let me explicitly point out that the main benefit of lambda expressions is that they remove a level of indirection from within your source code. Normally, you'd have to write a separate method, give that method a name, and then pass the name of that method where a delegate is required. The name gives you a way to refer to a body of code, and if you need to refer to the same body of code from multiple locations in your source code, then writing a method and giving it a name is a great way to go. However, if you need to have a body of code that is referred to only once within your source code, then a lambda expression allows you to put that code directly inline without having to assign it a name, thus increasing programmer productivity.

Note When C# 2.0 came out, it introduced a feature called *anonymous methods*. Like lambda expressions (introduced in C# 3.0), anonymous methods describes a syntax for creating anonymous functions. It is now recommended (in section 7.14 of the C# Language Specification) that developers use the newer lambda expression syntax rather than the older anonymous method syntax because the lambda expression syntax is more terse, making code easier to write, read, and maintain. Of course, Microsoft's C# compiler continues to support parsing both syntaxes for creating anonymous functions so that developers are not forced to modify any code that was originally written for C# 2.0. In this book, I will explain and use only the lambda expression syntax.

Syntactical Shortcut #3: No Need to Wrap Local Variables in a Class Manually to Pass Them to a Callback Method

I've already shown how the callback code can reference other members defined in the class. However, sometimes, you might like the callback code to reference local parameters or variables that exist in the defining method. Here's an interesting example.

```
internal sealed class AClass {
   public static void UsingLocalVariablesInTheCallbackCode(Int32 numToDo) {
      // Some local variables
      Int32[] squares = new Int32[numToDo];
      AutoResetEvent done = new AutoResetEvent(false);

      // Do a bunch of tasks on other threads
      for (Int32 n = 0; n < squares.Length; n++) {
         ThreadPool.QueueUserWorkItem(
            obj => {
               Int32 num = (Int32) obj;
```

```
            // This task would normally be more time consuming
            squares[num] = num * num;

            // If last task, let main thread continue running
            if (Interlocked.Decrement(ref numToDo) == 0)
               done.Set();
         },
         n);
      }

      // Wait for all the other threads to finish
      done.WaitOne();

      // Show the results
      for (Int32 n = 0; n < squares.Length; n++)
         Console.WriteLine("Index {0}, Square={1}", n, squares[n]);
   }
}
```

This example really shows off how easy C# makes implementing what used to be a pretty complex task. The preceding method defines one parameter, `numToDo`, and two local variables, `squares` and `done`. And the body of the lambda expression refers to these variables.

Now imagine that the code in the body of the lambda expression is placed in a separate method (as is required by the CLR). How would the values of the variables be passed to the separate method? The only way to do this is to define a new helper class that also defines a field for each value that you want passed to the callback code. In addition, the callback code would have to be defined as an instance method in this helper class. Then, the `UsingLocalVariablesInTheCallbackCode` method would have to construct an instance of the helper class, initialize the fields from the values in its local variables, and then construct the delegate object bound to the helper object/instance method.

Note When a lambda expression causes the compiler to generate a class with parameter/local variables turned into fields, the lifetime of the objects that the variables refer to are lengthened. Usually, a parameter/local variable goes out of scope at the last usage of the variable within a method. However, turning the variable into a field causes the field to keep the object that it refers to alive for the whole lifetime of the object containing the field. This is not a big deal in most applications, but it is something that you should be aware of.

This is very tedious and error-prone work, and, of course, the C# compiler does all this for you automatically. When you write the preceding code, it's as if the C# compiler rewrites your code so that it looks something like the following (comments inserted by me).

```
internal sealed class AClass {
   public static void UsingLocalVariablesInTheCallbackCode(Int32 numToDo) {

      // Some local variables
      WaitCallback callback1 = null;
```

```
      // Construct an instance of the helper class
      <>c__DisplayClass2 class1 = new <>c__DisplayClass2();

      // Initialize the helper class's fields
      class1.numToDo = numToDo;
      class1.squares = new Int32[class1.numToDo];
      class1.done = new AutoResetEvent(false);

      // Do a bunch of tasks on other threads
      for (Int32 n = 0; n < class1.squares.Length; n++) {
         if (callback1 == null) {
            // New up delegate object bound to the helper object and
            // its anonymous instance method
            callback1 = new WaitCallback(
               class1.<UsingLocalVariablesInTheCallbackCode>b__0);
         }

         ThreadPool.QueueUserWorkItem(callback1, n);
      }

      // Wait for all the other threads to finish
      class1.done.WaitOne();

      // Show the results
      for (Int32 n = 0; n < class1.squares.Length; n++)
         Console.WriteLine("Index {0}, Square={1}", n, class1.squares[n]);
   }

   // The helper class is given a strange name to avoid potential
   // conflicts and is private to forbid access from outside AClass
   [CompilerGenerated]
   private sealed class <>c__DisplayClass2 : Object {

      // One public field per local variable used in the callback code
      public Int32[] squares;
      public Int32 numToDo;
      public AutoResetEvent done;

      // public parameterless constructor
      public <>c__DisplayClass2 { }

      // Public instance method containing the callback code
      public void <UsingLocalVariablesInTheCallbackCode>b__0(Object obj) {
         Int32 num = (Int32) obj;
         squares[num] = num * num;
         if (Interlocked.Decrement(ref numToDo) == 0)
            done.Set();
      }
   }
}
```

Important Without a doubt, it doesn't take much for programmers to start abusing C#'s lambda expression feature. When I first started using lambda expressions, it definitely took me some time to get used to them. After all, the code that you write in a method is not actually inside that method, and this also can make debugging and single-stepping through the code a bit more challenging. In fact, I'm amazed at how well the Microsoft Visual Studio debugger actually handles stepping through lambda expressions in my source code.

I've set up a rule for myself: If I need my callback method to contain more than three lines of code, I will not use a lambda expression; instead, I'll write the method manually and assign it a name of my own creation. But, used judiciously, lambda expressions can greatly increase programmer productivity as well as the maintainability of your code. The following is some code in which using lambda expressions feels very natural. Without them, this code would be tedious to write, harder to read, and harder to maintain.

```
// Create and initialize a String array
String[] names = { "Jeff", "Kristin", "Aidan", "Grant" };

// Get just the names that have a lowercase 'a' in them.
Char charToFind = 'a';
names = Array.FindAll(names, name => name.IndexOf(charToFind) >= 0);

// Convert each string's characters to uppercase
names = Array.ConvertAll(names, name => name.ToUpper());

// Display the results
Array.ForEach(names, Console.WriteLine);
```

Delegates and Reflection

So far in this chapter, the use of delegates has required the developer to know up front the prototype of the method that is to be called back. For example, if `fb` is a variable that references a `Feedback` delegate (see this chapter's first program listing), to invoke the delegate, the code would look like the following.

```
fb(item);   // item is defined as Int32
```

As you can see, the developer must know when coding how many parameters the callback method requires and the types of those parameters. Fortunately, the developer almost always has this information, so writing code like the preceding code isn't a problem.

In some rare circumstances, however, the developer doesn't have this information at compile time. I showed an example of this in Chapter 11, "Events," when I discussed the `EventSet` type. In this example, a dictionary maintained a set of different delegate types. At run time, to raise an event, one of the delegates was looked up in the dictionary and invoked. At compile time, it wasn't possible to know exactly which delegate would be called and which parameters were necessary to pass to the delegate's callback method.

Fortunately, System.Reflection.MethodInfo offers a CreateDelegate method that allows you to create a delegate when you just don't have all the necessary information about the delegate at compile time. Here are the method overloads that MethodInfo defines.

```
public abstract class MethodInfo : MethodBase {
   // Construct a delegate wrapping a static method.
   public virtual Delegate CreateDelegate(Type delegateType);

   // Construct a delegate wrapping an instance method; target refers to the 'this' argument.
   public virtual Delegate CreateDelegate(Type delegateType, Object target);
}
```

After you've created the delegate, you can call it by using Delegate's DynamicInvoke method, which looks like the following.

```
public abstract class Delegate {
   // Invoke a delegate passing it parameters
   public Object DynamicInvoke(params Object[] args);
}
```

Using reflection APIs (discussed in Chapter 23, "Assembly Loading and Reflection"), you must first acquire a MethodInfo object referring to the method you want to create a delegate to. Then, you call the CreateDelegate method to have it construct a new object of a Delegate-derived type identified by the first parameter, delegateType. If the delegate wraps an instance method, you will also pass to CreateDelegate a target parameter indicating the object that should be passed as the this parameter to the instance method.

System.Delegate's DynamicInvoke method allows you to invoke a delegate object's callback method, passing a set of parameters that you determine at run time. When you call DynamicInvoke, it internally ensures that the parameters you pass are compatible with the parameters the callback method expects. If they're compatible, the callback method is called. If they're not, an ArgumentException is thrown. DynamicInvoke returns the object the callback method returned.

The following code shows how to use the CreateDelegate and DynamicInvoke methods.

```
using System;
using System.Reflection;
using System.IO;


// Here are some different delegate definitions
internal delegate Object TwoInt32s(Int32 n1, Int32 n2);
internal delegate Object OneString(String s1);


public static class DelegateReflection {
   public static void Main(String[] args) {
      if (args.Length < 2) {
         String usage =
            @"Usage:" +
            "{0} delType methodName [Arg1] [Arg2]" +
            "{0}   where delType must be TwoInt32s or OneString" +
```

```
      "{0}   if delType is TwoInt32s, methodName must be Add or Subtract" +
      "{0}   if delType is OneString, methodName must be NumChars or Reverse" +
      "{0}" +
      "{0}Examples:" +
      "{0}   TwoInt32s Add 123 321" +
      "{0}   TwoInt32s Subtract 123 321" +
      "{0}   OneString NumChars \"Hello there\"" +
      "{0}   OneString Reverse  \"Hello there\"";
   Console.WriteLine(usage, Environment.NewLine);
   return;
}

// Convert the delType argument to a delegate type
Type delType = Type.GetType(args[0]);
if (delType == null) {
   Console.WriteLine("Invalid delType argument: " + args[0]);
   return;
}

Delegate d;
try {
   // Convert the Arg1 argument to a method
   MethodInfo mi = typeof(DelegateReflection).GetTypeInfo().GetDeclaredMethod(args[1]);

   // Create a delegate object that wraps the static method
   d = mi.CreateDelegate(delType);
}
catch (ArgumentException) {
   Console.WriteLine("Invalid methodName argument: " + args[1]);
   return;
}

// Create an array that that will contain just the arguments
// to pass to the method via the delegate object
Object[] callbackArgs = new Object[args.Length - 2];

if (d.GetType() == typeof(TwoInt32s)) {
   try {
      // Convert the String arguments to Int32 arguments
      for (Int32 a = 2; a < args.Length; a++)
         callbackArgs[a - 2] = Int32.Parse(args[a]);
   }
   catch (FormatException) {
      Console.WriteLine("Parameters must be integers.");
      return;
   }
}

if (d.GetType() == typeof(OneString)) {
   // Just copy the String argument
   Array.Copy(args, 2, callbackArgs, 0, callbackArgs.Length);
}
```

```
      try {
         // Invoke the delegate and show the result
         Object result = d.DynamicInvoke(callbackArgs);
         Console.WriteLine("Result = " + result);
      }
      catch (TargetParameterCountException) {
         Console.WriteLine("Incorrect number of parameters specified.");
      }
   }


   // This callback method takes 2 Int32 arguments
   private static Object Add(Int32 n1, Int32 n2) {
      return n1 + n2;
   }

   // This callback method takes 2 Int32 arguments
   private static Object Subtract(Int32 n1, Int32 n2) {
      return n1 - n2;
   }

   // This callback method takes 1 String argument
   private static Object NumChars(String s1) {
      return s1.Length;
   }

   // This callback method takes 1 String argument
   private static Object Reverse(String s1) {
      return new String(s1.Reverse().ToArray());
   }
}
```