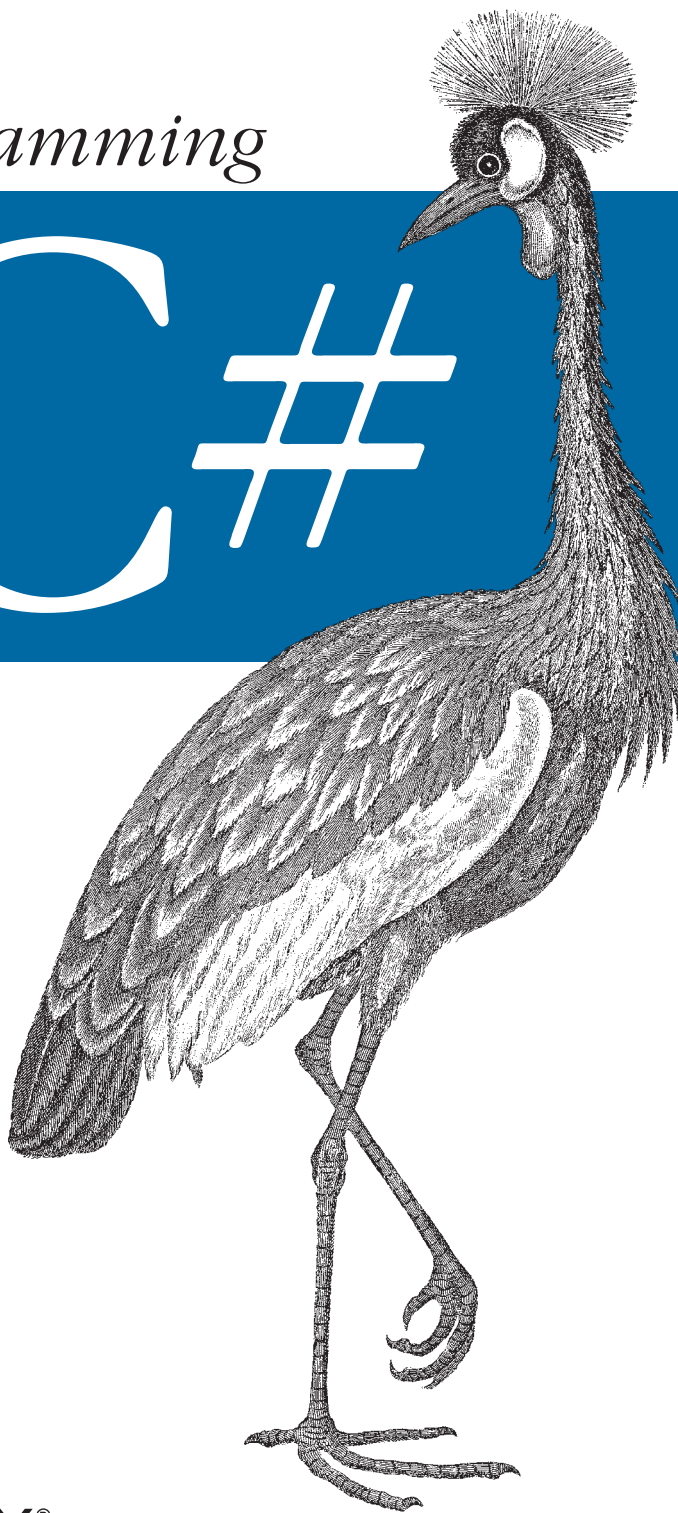


Building .NET Applications

4th Edition
Covers C# 2.0, .NET 2.0
& Visual Studio 2005

Programming

C#



O'REILLY®

Jesse Liberty

Delegates and Events

When a head of state dies, the president of the United States typically doesn't have time to attend the funeral personally. Instead, he dispatches a delegate. Often this delegate is the vice president, but sometimes the VP is unavailable and the president must send someone else, such as the secretary of state or even the first lady. He doesn't want to "hardwire" his delegated authority to a single person; he might delegate this responsibility to anyone who is able to execute the correct international protocol.

The president defines in advance what responsibility will be delegated (attend the funeral), what parameters will be passed (condolences, kind words), and what value he hopes to get back (good will). He then assigns a particular person to that delegated responsibility at "runtime" as the course of his presidency progresses.

In programming, you are often faced with situations where you need to execute a particular action, but you don't know in advance which method, or even which object, you'll want to call upon to execute it. For example, a button might know that it must notify some object when it is pushed, but it might not know which object or objects need to be notified. Instead of wiring the button to a particular object, you will connect the button to a *delegate* and then resolve that delegate to a particular method when the program executes.

In the early, dark, and primitive days of computing, a program would begin execution and then proceed through its steps until it completed. If the user was involved, the interaction was strictly controlled and limited to filling in fields.

Today's GUI programming model requires a different approach, known as *event-driven programming*. A modern program presents the user interface and waits for the user to take an action. The user might take many different actions, such as choosing among menu selections, pushing buttons, updating text fields, clicking icons, and so forth. Each action causes an event to be raised. Other events can be raised without direct user action, such as events that correspond to timer ticks of the internal clock, email being received, file-copy operations completing, etc.

An event is the encapsulation of the idea that “something happened” to which the program must respond. Events and delegates are tightly coupled concepts because flexible event handling requires that the response to the event be dispatched to the appropriate event handler. An event handler is typically implemented in C# via a delegate.

Delegates are also used as callbacks so that one class can say to another “do this work and when you’re done, let me know.”

Delegates

In C#, delegates are first-class objects, fully supported by the language. Technically, a delegate is a reference type used to encapsulate a method with a specific signature and return type.* You can encapsulate any matching method in that delegate.



In C++ and many other languages, you can to some degree accomplish this requirement with function pointers and pointers to member functions.

A delegate is created with the `delegate` keyword, followed by a return type and the signature of the methods that can be delegated to it, as in the following:

```
public delegate int WhichIsFirst(object obj1, object obj2);
```

This declaration defines a delegate named `WhichIsFirst`, which will encapsulate any method that takes two objects as parameters and that returns an `int`.

Once the delegate is defined, you can encapsulate a member method with that delegate by instantiating the delegate, passing in a method that matches the return type and signature. As an alternative, you can use anonymous methods as described later. In either case, the delegate can then be used to invoke that encapsulated method.

Using Delegates to Specify Methods at Runtime

Delegates *decouple* the class that declares the delegate from the class that uses the delegate. For example, suppose that you want to create a simple generic container class called a `Pair` that can hold and sort any two objects passed to it. You can't know in advance what kind of objects a `Pair` will hold, but by creating methods within those objects to which the sorting task can be delegated, you can delegate responsibility for determining their order to the objects themselves.

Different objects will sort differently (for example, a `Pair` of `Counter` objects might sort in numeric order, while a `Pair` of `Buttons` might sort alphabetically by their

* If the method is an instance method, the delegate encapsulates the target object as well.

name). As the author of the `Pair` class, you want the objects in the pair to have the responsibility of knowing which should be first and which should be second. To accomplish this, you will insist that the objects to be stored in the `Pair` must provide a method that tells you how to sort the objects.

You can define this requirement with interfaces, as well. Delegates are smaller and of finer granularity than interfaces. The `Pair` class doesn't need to implement an entire interface, it just needs to define the signature and return type of the method it wants to invoke. That is what delegates are for: they define the return type and signature of methods that can be invoked through the interface.

In this case, the `Pair` class will declare a delegate named `WhichIsFirst`. When the `Pair` needs to know how to order its objects, it will invoke the delegate passing in its two member objects as parameters. The responsibility for deciding which of the two objects comes first is delegated to the method encapsulated by the delegate:

```
public delegate Comparison
    WhichIsFirst( T obj1, T obj2 )
```

In this definition, `WhichIsFirst` is defined to encapsulate a method that takes two objects as parameters, and that returns an object of type `Comparison`. `Comparison` turns out to be an enumeration you will define:

```
public enum Comparison
{
    theFirstComesFirst = 1,
    theSecondComesFirst = 2
}
```

To test the delegate, you will create two classes, a `Dog` class and a `Student` class. `Dogs` and `Students` have little in common, except that they both implement methods that can be encapsulated by `WhichComesFirst`, and thus both `Dog` objects and `Student` objects are eligible to be held within `Pair` objects.

In the test program, you will create a couple of `Students` and a couple of `Dogs`, and store them each in a `Pair`. You will then create instances of `WhichIsFirst` to encapsulate their respective methods that will determine which `Student` or which `Dog` object should be first, and which second. Let's take this step by step.

You begin by creating a `Pair` constructor that takes two objects and stashes them away in a private array:

```
public Pair(
    T firstObject,
    T secondObject )
{
    thePair[0] = firstObject;
    thePair[1] = secondObject;
}
```

Next, you override `ToString()` to obtain the string value of the two objects:

```
public override string ToString()
{
    return thePair [0].ToString() + ", " +
           thePair[1].ToString();
}
```

You now have two objects in your `Pair` and you can print out their values. You're ready to sort them and print the results of the sort. You can't know in advance what kind of objects you will have, so you delegate the responsibility of deciding which object comes first in the sorted `Pair` to the objects themselves.

Both the `Dog` class and the `Student` class implement methods that can be encapsulated by `WhichIsFirst`. Any method that takes two objects and returns a `Comparison` can be encapsulated by this delegate at runtime.

You can now define the `Sort()` method for the `Pair` class:

```
public void Sort(WhichIsFirst theDelegatedFunc)
{
    if (theDelegatedFunc(thePair[0],thePair[1]) ==
        Comparison.theSecondComesFirst)
    {
        T temp = thePair[0];
        thePair[0] = thePair[1];
        thePair[1] = temp;
    }
}
```

This method takes a parameter: a delegate of type `WhichIsFirst` named `theDelegatedFunc`. The `Sort()` method delegates responsibility for deciding which of the two objects in the `Pair` comes first to the method encapsulated by that delegate. In the body of the `Sort()` method, it invokes the delegated method and examines the return value, which will be one of the two enumerated values of `Comparison`.

If the value returned is `theSecondComesFirst`, the objects within the pair are swapped; otherwise no action is taken.

This is analogous to how the other parameters work. If you had a method that took an `int` as a parameter:

```
int SomeMethod (int myParam){//...}
```

the parameter name is `myParam`, but you can pass in any `int` value or variable. Similarly, the parameter name in the delegate example is `theDelegatedFunc`, but you can pass in any method that meets the return value and signature defined by the delegate `WhichIsFirst`.

Imagine you are sorting `Students` by name. You write a method that returns `theFirstComesFirst` if the first student's name comes first, and `theSecondComesFirst` if the second student's name does. If you pass in "Amy, Beth" the method returns `theFirstComesFirst`, and if you pass in "Beth, Amy" it returns `theSecondComesFirst`. If

you get back `theSecondComesFirst`, the `Sort()` method reverses the items in its array, setting Amy to the first position and Beth to the second.

Now add one more method, `ReverseSort()`, which forces the items in the array into the reverse of their normal order:

```
public void ReverseSort(WhichIsFirst theDelegatedFunc)
{
    if (theDelegatedFunc(thePair[0], thePair[1]) ==
        Comparison.theFirstComesFirst)
    {
        T temp = thePair[0];
        thePair[0] = thePair[1];
        thePair[1] = temp;
    }
}
```

The logic here is identical to `Sort()`, except that this method performs the swap if the delegated method says that the first item comes first. Because the delegated function thinks the first item comes first, and this is a reverse sort, the result you want is for the second item to come first. This time if you pass in “Amy, Beth,” the delegated function returns `theFirstComesFirst` (i.e., Amy should come first), but because this is a *reverse* sort, it swaps the values, setting Beth first. This allows you to use the same delegated function as you used with `Sort()`, without forcing the object to support a function that returns the reverse sorted value.

Now all you need are some objects to sort. You’ll create two absurdly simple classes: `Student` and `Dog`. Assign `Student` objects a name at creation:

```
public class Student
{
    public Student(string name)
    {
        this.name = name;
    }
}
```

The `Student` class requires two methods: one to override `ToString()` and the other to be encapsulated as the delegated method.

`Student` must override `ToString()` so that the `ToString()` method in `Pair`, which invokes `ToString()` on the contained objects, will work properly; the implementation does nothing more than return the student’s name (which is already a string object):

```
public override string ToString()
{
    return name;
}
```

It must also implement a method to which `Pair.Sort()` can delegate the responsibility of determining which of two objects comes first:

```

return (String.Compare(s1.name, s2.name) < 0 ?
    Comparison.theFirstComesFirst :
    Comparison.theSecondComesFirst);

```

`String.Compare()` is a .NET Framework method on the `String` class that compares two strings and returns less than zero if the first is smaller, greater than zero if the second is smaller, and zero if they are the same. This method was discussed in some detail in Chapter 10. Notice that the logic here returns `theFirstComesFirst` only if the first string is smaller; if they are the same or the second is larger, this method returns `theSecondComesFirst`.

Notice that the `WhichStudentComesFirst()` method takes two objects as parameters and returns a `Comparison`. This qualifies it to be a `Pair.WhichIsFirst` delegated method, whose signature and return value it matches.

The second class is `Dog`. For our purposes, `Dog` objects will be sorted by weight, lighter dogs before heavier. Here's the complete declaration of `Dog`:

```

public class Dog
{
    public Dog(int weight)
    {
        this.weight=weight;
    }

    // dogs are ordered by weight
    public static Comparison WhichDogComesFirst(
        Object o1, Object o2)
    {
        Dog d1 = (Dog) o1;
        Dog d2 = (Dog) o2;
        return d1.weight > d2.weight ?
            Comparison.theSecondComesFirst :
            Comparison.theFirstComesFirst;
    }
    public override string ToString()
    {
        return weight.ToString();
    }
    private int weight;
}

```

The `Dog` class also overrides `ToString` and implements a static method with the correct signature for the delegate. Notice also that the `Dog` and `Student` delegate methods don't have the same name. They don't need to have the same name, as they will be assigned to the delegate dynamically at runtime.



You can call your delegated method names anything you like, but creating parallel names (for example, `WhichStudentComesFirst` and `WhichDogComesFirst`) makes the code easier to read, understand, and maintain.

Example 12-1 is the complete program, which illustrates how the delegate methods are invoked.

Example 12-1. Working with delegates

```
#region Using directives

using System;
using System.Collections.Generic;
using System.Text;

#endregion

namespace Delegates
{
    public enum Comparison
    {
        theFirstComesFirst = 1,
        theSecondComesFirst = 2
    }

    // a simple collection to hold 2 items
    public class Pair<T>
    {

        // private array to hold the two objects
        private T[] thePair = new T[2];

        // the delegate declaration
        public delegate Comparison
            WhichIsFirst( T obj1, T obj2 );

        // passed in constructor take two objects,
        // added in order received
        public Pair(
            T firstObject,
            T secondObject )
        {
            thePair[0] = firstObject;
            thePair[1] = secondObject;
        }

        // public method which orders the two objects
        // by whatever criteria the object likes!
        public void Sort(
            WhichIsFirst theDelegatedFunc )
        {
            if ( theDelegatedFunc( thePair[0], thePair[1] )
                == Comparison.theSecondComesFirst )
            {
                T temp = thePair[0];
                thePair[0] = thePair[1];
                thePair[1] = temp;
            }
        }
    }
}
```

Example 12-1. Working with delegates (continued)

```
    }
}

// public method which orders the two objects
// by the reverse of whatever criteria the object likes!
public void ReverseSort(
    WhichIsFirst theDelegatedFunc )
{
    if ( theDelegatedFunc( thePair[0], thePair[1] ) ==
        Comparison.theFirstComesFirst )
    {
        T temp = thePair[0];
        thePair[0] = thePair[1];
        thePair[1] = temp;
    }
}

// ask the two objects to give their string value
public override string ToString()
{
    return thePair[0].ToString() + ", "
        + thePair[1].ToString();
}
} // end class Pair

public class Dog
{
    private int weight;

    public Dog( int weight )
    {
        this.weight = weight;
    }

    // dogs are ordered by weight
    public static Comparison WhichDogComesFirst(
        Dog d1, Dog d2 )
    {
        return d1.weight > d2.weight ?
            Comparison.theSecondComesFirst :
            Comparison.theFirstComesFirst;
    }
    public override string ToString()
    {
        return weight.ToString();
    }
} // end class Dog

public class Student
{
    private string name;
```

Example 12-1. Working with delegates (continued)

```
public Student( string name )
{
    this.name = name;
}

// students are ordered alphabetically
public static Comparison
    WhichStudentComesFirst( Student s1, Student s2 )
{
    return ( String.Compare( s1.name, s2.name ) < 0 ?
        Comparison.theFirstComesFirst :
        Comparison.theSecondComesFirst );
}

public override string ToString()
{
    return name;
}
} // end class Student

public class Test
{
    public static void Main()
    {
        // create two students and two dogs
        // and add them to Pair objects
        Student Jesse = new Student( "Jesse" );
        Student Stacey = new Student( "Stacey" );
        Dog Milo = new Dog( 65 );
        Dog Fred = new Dog( 12 );

        Pair<Student> studentPair = new Pair<Student>( Jesse, Stacey );
        Pair<Dog> dogPair = new Pair<Dog>( Milo, Fred );
        Console.WriteLine( "studentPair\t\t\t: {0}",
            studentPair.ToString() );
        Console.WriteLine( "dogPair\t\t\t\t: {0}",
            dogPair.ToString() );

        // Instantiate the delegates
        Pair<Student>.WhichIsFirst theStudentDelegate =
            new Pair<Student>.WhichIsFirst(
                Student.WhichStudentComesFirst );

        Pair<Dog>.WhichIsFirst theDogDelegate =
            new Pair<Dog>.WhichIsFirst(
                Dog.WhichDogComesFirst );

        // sort using the delegates
        studentPair.Sort( theStudentDelegate );
        Console.WriteLine( "After Sort studentPair\t\t: {0}",
            studentPair.ToString() );
        studentPair.ReverseSort( theStudentDelegate );
    }
}
```

Example 12-1. Working with delegates (continued)

```
        Console.WriteLine( "After ReverseSort studentPair\t: {0}",
                           studentPair.ToString() );

        dogPair.Sort( theDogDelegate );
        Console.WriteLine( "After Sort dogPair\t\t: {0}",
                           dogPair.ToString() );
        dogPair.ReverseSort( theDogDelegate );
        Console.WriteLine( "After ReverseSort dogPair\t: {0}",
                           dogPair.ToString() );
    }
}
}
```

Output:

```
studentPair           : Jesse, Stacey
dogPair               : 65, 12
After Sort studentPair : Jesse, Stacey
After ReverseSort studentPair : Stacey, Jesse
After Sort dogPair    : 12, 65
After ReverseSort dogPair : 65, 12
```

The Test program creates two Student objects and two Dog objects and then adds them to Pair containers. The student constructor takes a string for the student's name and the dog constructor takes an int for the dog's weight:

```
Student Jesse = new Student( "Jesse" );
Student Stacey = new Student( "Stacey" );
Dog Milo = new Dog( 65 );
Dog Fred = new Dog( 12 );

Pair<Student> studentPair = new Pair<Student>( Jesse, Stacey );
Pair<Dog> dogPair = new Pair<Dog>( Milo, Fred );
Console.WriteLine( "studentPair\t\t\t: {0}",
                  studentPair.ToString() );
Console.WriteLine( "dogPair\t\t\t\t: {0}",
                  dogPair.ToString() );
```

It then prints the contents of the two Pair containers to see the order of the objects. The output looks like this:

```
studentPair           : Jesse, Stacey
dogPair               : 65, 12
```

As expected, the objects are in the order in which they were added to the Pair containers. We next instantiate two delegate objects:

```
Pair<Student>.WhichIsFirst theStudentDelegate =
    new Pair<Student>.WhichIsFirst(
        Student.WhichStudentComesFirst );

Pair<Dog>.WhichIsFirst theDogDelegate =
    new Pair<Dog>.WhichIsFirst(
        Dog.WhichDogComesFirst );
```

The first delegate, the `StudentDelegate`, is created by passing in the appropriate static method from the `Student` class. The second delegate, the `DogDelegate`, is passed a static method from the `Dog` class.

The delegates are now objects that can be passed to methods. You pass the delegates first to the `Sort()` method of the `Pair` object, and then to the `ReverseSort()` method. The results are printed to the console:

```
After Sort studentPair      : Jesse, Stacey
After ReverseSort studentPair : Stacey, Jesse
After Sort dogPair          : 12, 65
After ReverseSort dogPair    : 65, 12
```

Delegates and Instance Methods

In Example 12-1, the delegates encapsulate static methods, as in the following:

```
public static Comparison
    WhichStudentComesFirst(Student s1, Student s2)
```

The delegate is then instantiated using the class rather than an instance:

```
Pair<Student>.WhichIsFirst theStudentDelegate =
    new Pair<Student>.WhichIsFirst(
        Student.WhichStudentComesFirst);
```

You can just as easily encapsulate instance methods:

```
public Comparison
    WhichStudentComesFirst(Student s1, Student s2)
```

in which case you will instantiate the delegate by passing in the instance method as invoked through an instance of the class, rather than through the class itself:

```
Pair<Student>.WhichIsFirst theStudentDelegate =
    new Pair<Student>.WhichIsFirst(
        Jesse.WhichStudentComesFirst);
```

Static Delegates

One disadvantage of Example 12-1 is that it forces the calling class (in this case `Test`) to instantiate the delegates it needs to sort the objects in a `Pair`. It would be nice to get the delegate from the `Student` or `Dog` class itself. You can do this by giving each class its own static delegate. Thus, you can modify `Student` to add this:

```
public static readonly Pair<Student>.WhichIsFirst OrderStudents =
    new Pair<Student>.WhichIsFirst( Student.WhichStudentComesFirst );
```

This creates a static, read-only delegate field named `OrderStudents`.



Marking `OrderStudents` read-only denotes that once this static field is created, it isn't modified.

You can create a similar delegate within the `Dog` class:

```
public static readonly Pair<Dog>.WhichIsFirst OrderDogs =  
    new Pair<Dog>.WhichIsFirst( Dog.WhichDogComesFirst );
```

These are now static fields of their respective classes. Each is prepwired to the appropriate method within the class. You can invoke delegates without declaring a local delegate instance. You just pass in the static delegate of the class:

```
studentPair.Sort(Student.OrderStudents);  
Console.WriteLine("After Sort studentPair\t\t: {0}",  
    studentPair.ToString());  
studentPair.ReverseSort(Student.OrderStudents);  
Console.WriteLine("After ReverseSort studentPair\t: {0}",  
    studentPair.ToString());  
  
dogPair.Sort(Dog.OrderDogs);  
Console.WriteLine("After Sort dogPair\t\t: {0}",  
    dogPair.ToString());  
dogPair.ReverseSort(Dog.OrderDogs);  
Console.WriteLine("After ReverseSort dogPair\t: {0}",  
    dogPair.ToString());
```

The output after these changes is identical to Example 12-1.

Delegates as Properties

The problem with static delegates is that they must be instantiated, whether or not they are ever used, as with `Student` and `Dog` in Example 12-1. If you are creating hundreds of delegates you might consider implementing the static delegate fields as properties.

For `Student`, you take out the declaration:

```
public static readonly Pair<Student>.WhichIsFirst OrderStudents =  
    new Pair<Student>.WhichIsFirst( Student.WhichStudentComesFirst );
```

and replace it with:

```
public static Pair<Student>.WhichIsFirst OrderStudents  
{  
    get  
    {  
        return new Pair<Student>.WhichIsFirst( WhichStudentComesFirst );  
    }  
}
```

Similarly, you replace the static `Dog` field with:

```
public static Pair<Dog>.WhichIsFirst OrderDogs
{
    get
    {
        return new Pair<Dog>.WhichIsFirst( WhichDogComesFirst );
    }
}
```

The assignment of the delegates is unchanged:

```
studentPair.Sort(Student.OrderStudents);
dogPair.Sort(Dog.OrderDogs);
```

When the `OrderStudent` property is accessed, the delegate is created:

```
return new Pair.WhichIsFirst(WhichStudentComesFirst);
```

The key advantage is that the delegate is not created until it is requested. This allows the test class to determine when it needs a delegate, but still allows the details of the creation of the delegate to be the responsibility of the `Student` (or `Dog`) class.

Multicasting

At times, it is desirable to call two (or more) implementing methods through a single delegate. This becomes particularly important when handling events (discussed later in this chapter).

The goal is to have a single delegate that invokes more than one method. For example, when a button is pressed, you might want to take more than one action.

Two delegates can be combined with the addition operator (+). The result is a new multicast delegate that invokes both of the original implementing methods. For example, assuming `Writer` and `Logger` are delegates, the following line will combine them and produce a new multicast delegate named `myMulticastDelegate`:

```
myMulticastDelegate = Writer + Logger;
```

You can add delegates to a multicast delegate using the plus-equals (+=) operator. This operator adds the delegate on the right side of the operator to the multicast delegate on the left. For example, assuming `Transmitter` and `myMulticastDelegate` are delegates, the following line adds `Transmitter` to `myMulticastDelegate`:

```
myMulticastDelegate += Transmitter;
```

To see how multicast delegates are created and used, let's walk through a complete example. In Example 12-2, you will create a class called `MyClassWithDelegate` that defines a delegate that takes a string as a parameter and returns void:

```
public delegate void StringDelegate(string s);
```

You then define a class called `MyImplementingClass` that has three methods, all of which return void and take a string as a parameter: `WriteString`, `LogString`, and

TransmitString. The first writes the string to standard output, the second simulates writing to a log file, and the third simulates transmitting the string across the Internet. You instantiate the delegates to invoke the appropriate methods:

```
Writer("String passed to Writer\n");
Logger("String passed to Logger\n");
Transmitter("String passed to Transmitter\n");
```

To see how to combine delegates, you create another delegate instance:

```
MyClassWithDelegate.StringDelegate myMulticastDelegate;
```

and assign to it the result of “adding” two existing delegates:

```
myMulticastDelegate = Writer + Logger;
```

You add to this delegate an additional delegate using the += operator:

```
myMulticastDelegate += Transmitter;
```

Finally, you selectively remove delegates using the -= operator:

```
myMulticastDelegate -= Logger;
```

Example 12-2 shows how to combine delegates in this way.

Example 12-2. Combining delegates

#region Using directives

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

#endregion

```
namespace MulticastDelegates
{
    public class MyClassWithDelegate
    {
        // the delegate declaration
        public delegate void StringDelegate( string s );
    }

    public class MyImplementingClass
    {
        public static void WriteString( string s )
        {
            Console.WriteLine( "Writing string {0}", s );
        }

        public static void LogString( string s )
        {
            Console.WriteLine( "Logging string {0}", s );
        }
    }
}
```

Example 12-2. Combining delegates (continued)

```
public static void TransmitString( string s )
{
    Console.WriteLine( "Transmitting string {0}", s );
}

public class Test
{
    public static void Main()
    {
        // define three StringDelegate objects
        MyClassWithDelegate.StringDelegate
            Writer, Logger, Transmitter;

        // define another StringDelegate
        // to act as the multicast delegate
        MyClassWithDelegate.StringDelegate
            myMulticastDelegate;

        // Instantiate the first three delegates,
        // passing in methods to encapsulate
        Writer = new MyClassWithDelegate.StringDelegate(
            MyImplementingClass.WriteString );
        Logger = new MyClassWithDelegate.StringDelegate(
            MyImplementingClass.LogString );
        Transmitter =
            new MyClassWithDelegate.StringDelegate(
                MyImplementingClass.TransmitString );

        // Invoke the Writer delegate method
        Writer( "String passed to Writer\n" );

        // Invoke the Logger delegate method
        Logger( "String passed to Logger\n" );

        // Invoke the Transmitter delegate method
        Transmitter( "String passed to Transmitter\n" );

        // Tell the user you are about to combine
        // two delegates into the multicast delegate
        Console.WriteLine(
            "myMulticastDelegate = Writer + Logger" );

        // combine the two delegates, the result is
        // assigned to myMulticast Delegate
        myMulticastDelegate = Writer + Logger;

        // Call the delegated methods, two methods
        // will be invoked
        myMulticastDelegate(
            "First string passed to Collector" );
    }
}
```

Example 12-2. Combining delegates (continued)

```
// Tell the user you are about to add
// a third delegate to the multicast
Console.WriteLine(
    "\nmyMulticastDelegate += Transmitter" );

// add the third delegate
myMulticastDelegate += Transmitter;

// invoke the three delegated methods
myMulticastDelegate(
    "Second string passed to Collector" );

// tell the user you are about to remove
// the logger delegate
Console.WriteLine(
    "\nmyMulticastDelegate -= Logger" );

// remove the logger delegate
myMulticastDelegate -= Logger;

// invoke the two remaining
// delegated methods
myMulticastDelegate(
    "Third string passed to Collector" );
}
}
}
```

Output:

Writing string String passed to Writer

Logging string String passed to Logger

Transmitting string String passed to Transmitter

myMulticastDelegate = Writer + Logger

Writing string First string passed to Collector

Logging string First string passed to Collector

myMulticastDelegate += Transmitter

Writing string Second string passed to Collector

Logging string Second string passed to Collector

Transmitting string Second string passed to Collector

myMulticastDelegate -= Logger

Writing string Third string passed to Collector

Transmitting string Third string passed to Collector

In the Test portion of Example 12-2, the delegate instances are defined and the first three (Writer, Logger, and Transmitter) are invoked. The fourth delegate, myMulticastDelegate, is then assigned the combination of the first two, and it is

invoked, causing both delegated methods to be called. The third delegate is added, and when `myMulticastDelegate` is invoked, all three delegated methods are called. Finally, `Logger` is removed, and when `myMulticastDelegate` is invoked, only the two remaining methods are called.

The power of multicast delegates is best understood in terms of events, discussed in the next section. When an event such as a button press occurs, an associated multicast delegate can invoke a series of event handler methods that will respond to the event.

Events

GUIs, such as Microsoft Windows and web browsers, require that programs respond to *events*. An event might be a button push, a menu selection, the completion of a file transfer, and so forth. In short, something happens and you must respond to it. You can't predict the order in which events will arise. The system is quiescent until the event, and then springs into action to handle it.

In a GUI environment, any number of widgets can *raise* an event. For example, when you click a button, it might raise the `Click` event. When you add to a drop-down list, it might raise a `ListChanged` event.

Other classes will be interested in responding to these events. How they respond is not of interest to the class raising the event. The button says, "I was clicked," and the responding classes react appropriately.

Publishing and Subscribing

In C#, any object can *publish* a set of events to which other classes can *subscribe*. When the publishing class raises an event, all the subscribed classes are notified. With this mechanism, your object can say, "Here are things I can notify you about," and other classes might sign up, saying, "Yes, let me know when that happens." For example, a button might notify any number of interested observers when it is clicked. The button is called the *publisher* because the button publishes the `Click` event and the other classes are the *subscribers* because they subscribe to the `Click` event.



This design implements the Publish/Subscribe (Observer) Pattern described in the seminal work *Design Patterns* (Addison Wesley). Gamma describes the intent of this pattern: "Define a one to many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."

Note that the publishing class doesn't know or care who (if anyone) subscribes; it just raises the event. Who responds to that event, and how they respond, isn't the concern of the publishing class.

As a second example, a `Clock` might notify interested classes whenever the time changes by one second. The `Clock` class could itself be responsible for the `User`

Interface representation of the time, instead of raising an event, so why bother with the indirection of using delegates? The advantage of the publish/subscribe idiom is that the Clock class need not know how its information will be used; the monitoring of the time is thus decoupled from the representation of that information. In addition, any number of classes can be notified when an event is raised. The subscribing classes don't need to know how the Clock works, and the Clock doesn't need to know what they are going to do in response to the event.

The publisher and the subscribers are decoupled by the delegate. This is highly desirable; it makes for more flexible and robust code. The Clock can change how it detects time without breaking any of the subscribing classes. The subscribing classes can change how they respond to time changes without breaking the Clock. The two classes spin independently of one another, and that makes for code that is easier to maintain.

Events and Delegates

Events in C# are implemented with delegates. The publishing class defines a delegate. The subscribing class does two things: first it creates a method that matches the signature of the delegate, and then it creates an instance of that delegate type encapsulating that method. When the event is raised, the subscribing class's methods are invoked through the delegate.

A method that handles an event is called an *event handler*. You can declare your event handlers as you would any other delegate.

By convention, event handlers in the .NET Framework return void and take two parameters. The first parameter is the "source" of the event (that is, the publishing object). The second parameter is an object derived from EventArgs. It is recommended that your event handlers follow this design pattern.



VB6 programmers take note: C# doesn't put restrictions on the names of the methods that handle events. Also, the .NET implementation of the publish/subscribe model lets you have a single method that subscribes to multiple events.

EventArgs is the base class for all event data. Other than its constructor, the EventArgs class inherits all its methods from Object, though it does add a public static field named empty, which represents an event with no state (to allow for the efficient use of events with no state). The EventArgs derived class contains information about the event.

Suppose you want to create a Clock class that uses delegates to notify potential subscribers whenever the local time changes value by one second. Call this delegate SecondChangeHandler.

The declaration for the `SecondChangeHandler` delegate is:

```
public delegate void SecondChangeHandler(
    object clock,
    TimeInfoEventArgs timeInformation
);
```

This delegate will encapsulate any method that returns `void` and that takes two parameters. The first parameter is an object that represents the clock (the object raising the event), and the second parameter is an object of type `TimeInfoEventArgs` that will contain useful information for anyone interested in this event. `TimeInfoEventArgs` is defined as follows:

```
public class TimeInfoEventArgs : EventArgs
{
    public TimeInfoEventArgs(int hour, int minute, int second)
    {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }
    public readonly int hour;
    public readonly int minute;
    public readonly int second;
}
```

The `TimeInfoEventArgs` object will have information about the current hour, minute, and second. It defines a constructor and three public, read-only integer variables.

In addition to its delegate, a `Clock` has three member variables—hour, minute, and second—as well as a single method, `Run()`:

```
public void Run()
{
    for(;;)
    {
        // sleep 10 milliseconds
        Thread.Sleep(10);

        // get the current time
        System.DateTime dt = System.DateTime.Now;

        // if the second has changed
        // notify the subscribers
        if (dt.Second != second)
        {
            // create the TimeInfoEventArgs object
            // to pass to the subscriber
            TimeInfoEventArgs timeInformation =
                new TimeInfoEventArgs(
                    dt.Hour, dt.Minute, dt.Second);

            // if anyone has subscribed, notify them
            if (OnSecondChange != null)
```

```

        {
            OnSecondChange(this,timeInformation);
        }
    }
    // update the state
    this.second = dt.Second;
    this.minute = dt.Minute;
    this.hour = dt.Hour;
}
}

```

Run() creates an infinite for loop that periodically checks the system time. If the time has changed from the Clock object's current time, it notifies all its subscribers and then updates its own state.

The first step is to sleep for 10 milliseconds:

```
Thread.Sleep(10);
```

This makes use of a static method of the Thread class from the System.Threading namespace, which will be covered in some detail in Chapter 20. The call to Sleep() prevents the loop from running so tightly that little else on the computer gets done.

After sleeping for 10 milliseconds, the method checks the current time:

```
System.DateTime dt = System.DateTime.Now;
```

About every 100 times it checks, the second will have incremented. The method notices that change and notifies its subscribers. To do so, it first creates a new TimeInfoEventArgs object:

```

if (dt.Second != second)
{
    // create the TimeInfoEventArgs object
    // to pass to the subscriber
    TimeInfoEventArgs timeInformation =
        new TimeInfoEventArgs(dt.Hour,dt.Minute,dt.Second);
}

```

It then notifies the subscribers by firing the OnSecondChange event:

```

// if anyone has subscribed, notify them
if (OnSecondChange != null)
{
    OnSecondChange(this,timeInformation);
}
}

```

If an event has no subscribers registered, it evaluates to null. The preceding test checks that the value isn't null, ensuring that there are subscribers before calling OnSecondChange.

Remember that OnSecondChange takes two arguments: the source of the event and the object derived from EventArgs. In the snippet, you see that the clock's this reference is passed because the clock is the source of the event. The second parameter is the TimeInfoEventArgs object, timeInformation, created on the line above.

Raising the event invokes whatever methods have been registered with the Clock class through the delegate. We'll examine this in a moment.

Once the event is raised, update the state of the Clock class:

```
this.second = dt.Second;  
this.minute = dt.Minute;  
this.hour = dt.Hour;
```



No attempt has been made to make this code thread-safe. Thread safety and synchronization are discussed in Chapter 20.

All that is left is to create classes that can subscribe to this event. You create two. First is the DisplayClock class. The job of DisplayClock isn't to keep track of time, but rather, to display the current time to the console.

The example simplifies this class down to two methods. The first is a helper method named Subscribe() that subscribes to the clock's OnSecondChange delegate. The second method is the event handler TimeHasChanged():

```
public class DisplayClock  
{  
    public void Subscribe(Clock theClock)  
    {  
        theClock.OnSecondChange +=  
            new Clock.SecondChangeHandler(TimeHasChanged);  
    }  
  
    public void TimeHasChanged(  
        object theClock, TimeInfoEventArgs ti)  
    {  
        Console.WriteLine("Current Time: {0}:{1}:{2}",  
            ti.hour.ToString(),  
            ti.minute.ToString(),  
            ti.second.ToString());  
    }  
}
```

When the first method, Subscribe(), is invoked, it creates a new SecondChangeHandler delegate, passing in its event handler method, TimeHasChanged(). It then registers that delegate with the OnSecondChange event of Clock.

Now create a second class that also responds to this event, LogCurrentTime. This class normally logs the event to a file, but for our demonstration purposes, it logs to the standard console:

```
public class LogCurrentTime  
{  
    public void Subscribe(Clock theClock)  
    {  
        theClock.OnSecondChange +=
```

```

        new Clock.SecondChangeHandler(WriteLogEntry);
    }

    // This method should write to a file.
    // We write to the console to see the effect.
    // This object keeps no state.
    public void WriteLogEntry(
        object theClock, TimeInfoEventArgs ti)
    {
        Console.WriteLine("Logging to file: {0}:{1}:{2}",
            ti.hour.ToString(),
            ti.minute.ToString(),
            ti.second.ToString());
    }
}

```

Although in this example these two classes are very similar, in a production program any number of disparate classes might subscribe to an event.

All that remains is to create a Clock class, create the DisplayClock class, and tell it to subscribe to the event. You then create a LogCurrentTime class and tell it to subscribe as well. Finally, tell the Clock to run. All this is shown in Example 12-3 (you need to press Ctrl-C to terminate this application).

Example 12-3. Implementing events with delegates

```

#region Using directives

using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

#endregion

namespace EventsWithDelegates
{
    // a class to hold the information about the event
    // in this case it will hold only information
    // available in the clock class, but could hold
    // additional state information
    public class TimeInfoEventArgs : EventArgs
    {
        public TimeInfoEventArgs( int hour, int minute, int second )
        {
            this.hour = hour;
            this.minute = minute;
            this.second = second;
        }
        public readonly int hour;
        public readonly int minute;
        public readonly int second;
    }
}

```

Example 12-3. Implementing events with delegates (continued)

```
// our subject -- it is this class that other classes
// will observe. This class publishes one delegate:
// OnSecondChange.
public class Clock
{
    private int hour;
    private int minute;
    private int second;

    // the delegate the subscribers must implement
    public delegate void SecondChangeHandler
        (
            object clock,
            TimeInfoEventArgs timeInformation
        );

    // an instance of the delegate
    public SecondChangeHandler OnSecondChange;

    // set the clock running
    // it will raise an event for each new second
    public void Run()
    {
        for ( ; ; )
        {
            // sleep 10 milliseconds
            Thread.Sleep( 10 );

            // get the current time
            System.DateTime dt = System.DateTime.Now;

            // if the second has changed
            // notify the subscribers
            if ( dt.Second != second )
            {
                // create the TimeInfoEventArgs object
                // to pass to the subscriber
                TimeInfoEventArgs timeInformation =
                    new TimeInfoEventArgs(
                        dt.Hour, dt.Minute, dt.Second );

                // if anyone has subscribed, notify them
                if ( OnSecondChange != null )
                {
                    OnSecondChange(
                        this, timeInformation );
                }
            }

            // update the state
            this.second = dt.Second;
        }
    }
}
```

Example 12-3. Implementing events with delegates (continued)

```
        this.minute = dt.Minute;
        this.hour = dt.Hour;
    }
}

// an observer. DisplayClock subscribes to the
// clock's events. The job of DisplayClock is
// to display the current time
public class DisplayClock
{
    // given a clock, subscribe to
    // its SecondChangeHandler event
    public void Subscribe( Clock theClock )
    {
        theClock.OnSecondChange +=
            new Clock.SecondChangeHandler( TimeHasChanged );
    }

    // the method that implements the
    // delegated functionality
    public void TimeHasChanged(
        object theClock, TimeInfoEventArgs ti )
    {
        Console.WriteLine( "Current Time: {0}:{1}:{2}",
            ti.hour.ToString(),
            ti.minute.ToString(),
            ti.second.ToString() );
    }
}

// a second subscriber whose job is to write to a file
public class LogCurrentTime
{
    public void Subscribe( Clock theClock )
    {
        theClock.OnSecondChange +=
            new Clock.SecondChangeHandler( WriteLogEntry );
    }

    // This method should write to a file.
    // We write to the console to see the effect.
    // This object keeps no state.
    public void WriteLogEntry(
        object theClock, TimeInfoEventArgs ti )
    {
        Console.WriteLine( "Logging to file: {0}:{1}:{2}",
            ti.hour.ToString(),
            ti.minute.ToString(),
            ti.second.ToString() );
    }
}
}
```

Example 12-3. Implementing events with delegates (continued)

```
public class Test
{
    public static void Main()
    {
        // create a new clock
        Clock theClock = new Clock();

        // create the display and tell it to
        // subscribe to the clock just created
        DisplayClock dc = new DisplayClock();
        dc.Subscribe( theClock );

        // create a Log object and tell it
        // to subscribe to the clock
        LogCurrentTime lct = new LogCurrentTime();
        lct.Subscribe( theClock );

        // Get the clock started
        theClock.Run();
    }
}
```

Output:

```
Current Time: 14:53:56
Logging to file: 14:53:56
Current Time: 14:53:57
Logging to file: 14:53:57
Current Time: 14:53:58
Logging to file: 14:53:58
Current Time: 14:53:59
Logging to file: 14:53:59
Current Time: 14:54:0
Logging to file: 14:54:0
```

The net effect of this code is to create two classes, `DisplayClock` and `LogCurrentTime`, both of which subscribe to a third class' event (`Clock.OnSecondChange`).

`OnSecondChange` is a multicast delegate field, initially referring to nothing. In time it refers to a single delegate, and then later to multiple delegates. When the observer classes wish to be notified, they create an instance of the delegate and then add these delegates to `OnSecondChange`. For example, in `DisplayClock`'s `Subscribe()` method, you see this line of code:

```
theClock.OnSecondChange +=
    new Clock.SecondChangeHandler(TimeHasChanged);
```

It turns out that the `LogCurrentTime` class also wants to be notified. In its `Subscribe()` method is very similar code:

```
public void Subscribe(Clock theClock)
{
```

```

    theClock.OnSecondChange +=
        new Clock.SecondChangeHandler(WriteLogEntry);
}

```

Solving Delegate Problems with Events

There is a problem with Example 12-3, however. What if the `LogCurrentTime` class was not so considerate, and it used the assignment operator (`=`) rather than the subscribe operator (`+=`), as in the following:

```

public void Subscribe(Clock theClock)
{
    theClock.OnSecondChange =
        new Clock.SecondChangeHandler(WriteLogEntry);
}

```

If you make that one tiny change to the example, you'll find that the `Logger()` method is called, but the `DisplayClock` method is *not* called. The assignment operator *replaced* the delegate held in the `OnSecondChange` multicast delegate. This isn't good.

A second problem is that other methods can call `SecondChangeHandler` directly. For example, you might add the following code to the `Main()` method of your `Test` class:

```

Console.WriteLine("Calling the method directly!");
System.DateTime dt = System.DateTime.Now.AddHours(2);

TimeInfoEventArgs timeInformation =
    new TimeInfoEventArgs(
        dt.Hour, dt.Minute, dt.Second);

theClock.OnSecondChange(theClock, timeInformation);

```

Here `Main()` has created its own `TimeInfoEventArgs` object and invoked `OnSecondChange` directly. This runs fine, even though it is not what the designer of the `Clock` class intended. Here is the output:

```

Calling the method directly!
Current Time: 18:36:7
Logging to file: 18:36:7
Current Time: 16:36:7
Logging to file: 16:36:7

```

The problem is that the designer of the `Clock` class intended the methods encapsulated by the delegate to be invoked only when the event is fired. Here `Main()` has gone around through the back door and invoked those methods itself. What is more, it has passed in bogus data (passing in a time construct set to two hours into the future!).

How can you, as the designer of the `Clock` class, ensure that no one calls the delegated method directly? You can make the delegate private, but then it won't be possible for clients to register with your delegate at all. What's needed is a way to say,

“This delegate is designed for event handling: you may subscribe and unsubscribe, but you may not invoke it directly.”

The event Keyword

The solution to this dilemma is to use the event keyword. The event keyword indicates to the compiler that the delegate can be invoked only by the defining class, and that other classes can only subscribe to and unsubscribe from the delegate using the appropriate += and -= operators, respectively.

To fix your program, change your definition of `OnSecondChange` from:

```
public SecondChangeHandler OnSecondChange;
```

to the following:

```
public event SecondChangeHandler OnSecondChange;
```

Adding the event keyword fixes both problems. Classes can no longer attempt to subscribe to the event using the assignment operator (=), as they could previously, nor can they invoke the event directly, as was done in `Main()` in the preceding example. Either of these attempts will now generate a compile error:

```
The event 'Programming_CSharp.Clock.OnSecondChange' can only appear on  
the left hand side of += or -= (except when used from within the type 'Programming_  
CSharp.Clock')
```

There are two ways of looking at `OnSecondChange` now that you’ve modified it. In one sense, it is simply a delegate instance to which you’ve restricted access using the keyword `event`. In another, more important sense, `OnSecondChange` is an event, implemented by a delegate of type `SecondChangeHandler`. These two statements mean the same thing, but the latter is a more object-oriented way of looking at it, and better reflects the intent of this keyword: to create an event that your object can raise, and to which other objects can respond.

The complete source, modified to use the event rather than the unrestricted delegate, is shown in Example 12-4.

Example 12-4. Using the event keyword

```
#region Using directives

using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

#endregion

namespace EventKeyword
{
```

Example 12-4. Using the event keyword (continued)

```
// a class to hold the information about the event
// in this case it will hold only information
// available in the clock class, but could hold
// additional state information
public class TimeInfoEventArgs : EventArgs
{
    public readonly int hour;
    public readonly int minute;
    public readonly int second;
    public TimeInfoEventArgs(int hour, int minute, int second)
    {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }
}

// our subject -- it is this class that other classes
// will observe. This class publishes one event:
// OnSecondChange. The observers subscribe to that event
public class Clock
{
    private int hour;
    private int minute;
    private int second;

    // the delegate the subscribers must implement
    public delegate void SecondChangeHandler
    (
        object clock,
        TimeInfoEventArgs timeInformation
    );

    // the keyword event controls access to the delegate
    public event SecondChangeHandler OnSecondChange;

    // set the clock running
    // it will raise an event for each new second
    public void Run()
    {
        for(;;)
        {
            // sleep 10 milliseconds
            Thread.Sleep(10);

            // get the current time
            System.DateTime dt = System.DateTime.Now;

            // if the second has changed
            // notify the subscribers
            if (dt.Second != second)
```

Example 12-4. Using the event keyword (continued)

```
{
    // create the TimeInfoEventArgs object
    // to pass to the subscriber
    TimeInfoEventArgs timeInformation =
        new TimeInfoEventArgs(
            dt.Hour,dt.Minute,dt.Second);

    // if anyone has subscribed, notify them
    if (OnSecondChange != null)
    {
        OnSecondChange(
            this,timeInformation);
    }
}

// update the state
this.second = dt.Second;
this.minute = dt.Minute;
this.hour = dt.Hour;
}
}

// an observer. DisplayClock subscribes to the
// clock's events. The job of DisplayClock is
// to display the current time
public class DisplayClock
{
    // given a clock, subscribe to
    // its SecondChangeHandler event
    public void Subscribe(Clock theClock)
    {
        theClock.OnSecondChange +=
            new Clock.SecondChangeHandler(TimeHasChanged);
    }

    // the method that implements the
    // delegated functionality
    public void TimeHasChanged(
        object theClock, TimeInfoEventArgs ti)
    {
        Console.WriteLine("Current Time: {0}:{1}:{2}",
            ti.hour.ToString(),
            ti.minute.ToString(),
            ti.second.ToString());
    }
}

// a second subscriber whose job is to write to a file
public class LogCurrentTime
{
    public void Subscribe(Clock theClock)
```

Example 12-4. Using the event keyword (continued)

```
{
    theClock.OnSecondChange +=
        new Clock.SecondChangeHandler(WriteLogEntry);
}

// This method should write to a file.
// We write to the console to see the effect.
// This object keeps no state.
public void WriteLogEntry(
    object theClock, TimeInfoEventArgs ti)
{
    Console.WriteLine("Logging to file: {0}:{1}:{2}",
        ti.hour.ToString(),
        ti.minute.ToString(),
        ti.second.ToString());
}
}

public class Test
{
    public static void Main()
    {
        // create a new clock
        Clock theClock = new Clock();

        // create the display and tell it to
        // subscribe to the clock just created
        DisplayClock dc = new DisplayClock();
        dc.Subscribe( theClock );

        // create a Log object and tell it
        // to subscribe to the clock
        LogCurrentTime lct = new LogCurrentTime();
        lct.Subscribe( theClock );

        // Get the clock started
        theClock.Run();
    }
}
}
```

Using Anonymous Methods

In the previous example, you subscribed to the event by invoking a new instance of the delegate, passing in the name of a method that implements the event:

```
theClock.OnSecondChange +=
    new Clock.SecondChangeHandler(TimeHasChanged);
```



You can also assign this delegate by writing the shortened version:
`theClock.OnSecondChange += TimeHasChanged`

Later in the code, you must define `TimeHasChanged` as a method that matches the signature of the `SecondChangeHandler` delegate:

```
public void TimeHasChanged(  
    object theClock, TimeInfoEventArgs ti)  
{  
    Console.WriteLine("Current Time: {0}:{1}:{2}",  
        ti.hour.ToString(),  
        ti.minute.ToString(),  
        ti.second.ToString());  
}
```

Anonymous methods allow you to pass a code block rather than the name of the method. This can make for more efficient and easier-to-maintain code, and the anonymous method has access to the variables in the scope in which they are defined:

```
clock.OnSecondChange += delegate( object theClock, TimeInfoEventArgs ti )  
{  
    Console.WriteLine( "Current Time: {0}:{1}:{2}",  
        ti.hour.ToString(),  
        ti.minute.ToString(),  
        ti.second.ToString() );  
};
```

Notice that instead of registering an instance of a delegate, you use the keyword `delegate`, followed by the parameters that would be passed to your method, followed by the body of your method encased in braces and terminated by a semicolon.

This “method” has no name, hence it is anonymous. You can invoke the method only through the delegate; but that is exactly what you want.

Retrieving Values from Multicast Delegates

In most situations, the methods you’ll encapsulate with a multicast delegate will return `void`. In fact, the most common use of multicast delegates is with events, and you will remember that by convention, all events are implemented by delegates that encapsulate methods that return `void` (and also take two parameters: the sender and an `EventArgs` object).

It is possible, however, to create multicast delegates for methods that don’t return `void`. In the next example, you will create a very simple test class with a delegate that encapsulates any method that takes no parameters but returns an integer:

```
public class ClassWithDelegate  
{  
    public delegate int DelegateThatReturnsInt();  
    public DelegateThatReturnsInt theDelegate;
```

To test this, you implement two classes that subscribe to your delegate. The first encapsulates a method that increments a counter and returns that value as an integer:

```
public class FirstSubscriber
{
    private int myCounter = 0;

    public void Subscribe(ClassWithDelegate theClassWithDelegate)
    {
        theClassWithDelegate.theDelegate +=
            new ClassWithDelegate.DelegateThatReturnsInt(DisplayCounter);
    }

    public int DisplayCounter()
    {
        return ++myCounter;
    }
}
```

The second class also maintains a counter, but its delegated method doubles the counter and returns that doubled value:

```
public class SecondSubscriber
{
    private int myCounter = 0;

    public void Subscribe(ClassWithDelegate theClassWithDelegate)
    {
        theClassWithDelegate.theDelegate +=
            new ClassWithDelegate.DelegateThatReturnsInt(Doubler);
    }

    public int Doubler()
    {
        return myCounter += 2;
    }
}
```

When you fire this delegate, each encapsulated method is called in turn, and each returns a value:

```
int result = theDelegate();
Console.WriteLine(
    "Delegates fired! Returned result: {0}",
    result);
```

The problem is that as each method returns its value, it overwrites the value assigned to result. The output looks like this:

```
Delegates fired! Returned result: 2
Delegates fired! Returned result: 4
Delegates fired! Returned result: 6
Delegates fired! Returned result: 8
Delegates fired! Returned result: 10
```

The first method, `DisplayCounter()` (which was called by `FirstSubscriber`), returned the values 1,2,3,4,5, but these values were overwritten by the values returned by the second method.

Your goal is to display the result of each method invocation in turn. To do so, you must take over the responsibility of invoking the methods encapsulated by your multicast delegate. You do so by obtaining the invocation list from your delegate and explicitly invoking each encapsulated method in turn:

```
foreach (
    DelegateThatReturnsInt del in
        theDelegate.GetInvocationList() )
{
    int result = del();
    Console.WriteLine(
        "Delegates fired! Returned result: {0}",
        result);
}
Console.WriteLine();
```

This time, `result` is assigned the value of each invocation, and that value is displayed before invoking the next method. The output reflects this change:

```
Delegates fired! Returned result: 1
Delegates fired! Returned result: 2

Delegates fired! Returned result: 2
Delegates fired! Returned result: 4

Delegates fired! Returned result: 3
Delegates fired! Returned result: 6

Delegates fired! Returned result: 4
Delegates fired! Returned result: 8

Delegates fired! Returned result: 5
Delegates fired! Returned result: 10
```

The first delegated method is counting up (1,2,3,4,5) while the second is doubling (2,4,6,8,10). The complete source is shown in Example 12-5.

Example 12-5. Invoking delegated methods manually

```
#region Using directives

using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

#endregion

namespace InvokingDelegatedMethodsManually
{
```

Example 12-5. Invoking delegated methods manually (continued)

```
public class ClassWithDelegate
{
    // a multicast delegate that encapsulates a method
    // that returns an int
    public delegate int DelegateThatReturnsInt();
    public DelegateThatReturnsInt theDelegate;

    public void Run()
    {
        for ( ; ; )
        {
            // sleep for a half second
            Thread.Sleep( 500 );

            if ( theDelegate != null )
            {
                // explicitly invoke each delegated method
                foreach (
                    DelegateThatReturnsInt del in
                    theDelegate.GetInvocationList() )
                {
                    int result = del();
                    Console.WriteLine(
                        "Delegates fired! Returned result: {0}",
                        result );
                } // end foreach
                Console.WriteLine();
            } // end if
        } // end for ;;
    } // end run
} // end class

public class FirstSubscriber
{
    private int myCounter = 0;

    public void Subscribe( ClassWithDelegate theClassWithDelegate )
    {
        theClassWithDelegate.theDelegate +=
            new ClassWithDelegate.DelegateThatReturnsInt( DisplayCounter );
    }

    public int DisplayCounter()
    {
        return ++myCounter;
    }
}

public class SecondSubscriber
{
    private int myCounter = 0;
```

Example 12-5. Invoking delegated methods manually (continued)

```
public void Subscribe( ClassWithDelegate theClassWithDelegate )
{
    theClassWithDelegate.theDelegate +=
        new ClassWithDelegate.DelegateThatReturnsInt( Doubler );
}

public int Doubler()
{
    return myCounter += 2;
}

public class Test
{
    public static void Main()
    {
        ClassWithDelegate theClassWithDelegate =
            new ClassWithDelegate();

        FirstSubscriber fs = new FirstSubscriber();
        fs.Subscribe( theClassWithDelegate );

        SecondSubscriber ss = new SecondSubscriber();
        ss.Subscribe( theClassWithDelegate );

        theClassWithDelegate.Run();
    }
}
```

Invoking Events Asynchronously

It may turn out that the event handlers take longer than you like to respond to the event. In that case, it may take a while to notify later handlers, while you wait for results from earlier handlers. For example, suppose the `DisplayCounter()` method in `FirstSubscriber` needs to do a lot of work to compute the return result. This would create a delay before `SecondSubscriber` was notified of the event. You can simulate this by adding a few lines to `DisplayCounter`:

```
public int DisplayCounter()
{
    Console.WriteLine("Busy in DisplayCounter...");
    Thread.Sleep(4000);
    Console.WriteLine("Done with work in DisplayCounter...");
    return ++myCounter;
}
```

When you run the program, you can see the four-second delay each time `FirstSubscriber` is notified. An alternative to invoking each method through the delegates (as shown earlier) is to call the `BeginInvoke()` method on each delegate. This will

cause the methods to be invoked asynchronously, and you can get on with your work, without waiting for the method you invoke to return.

Unlike `Invoke()`, `BeginInvoke()` returns immediately. It creates a separate thread in which its own work is done.* (For more information about threads, see Chapter 20.)

This presents a problem, however, since you do want to get the results from the methods you invoke. You have two choices. First, you can constantly poll each delegated method, asking if it has a result yet. This would be like asking your assistant to do some work for you and then telephoning every five seconds saying, “Is it done yet?” (a waste of everybody’s time). What you want to do is to turn to your assistant and say, “Do this work, and call me when you have a result.”

Callback Methods

You accomplish this goal of delegating work and being called back when it is done with a callback, which you implement with (surprise!) a delegate. The .NET Framework provides a callback mechanism by defining the `AsyncCallback` delegate:

```
[Serializable]
public delegate void AsyncCallback(
    IAsyncResult iar
);
```

The attribute (`Serializable`) is covered in Chapter 18. You can see here, however, that `AsyncCallback` is a delegate for a method that returns `void` and takes a single argument, an object of type `IAsyncResult`. This interface is defined by the Framework, and the CLR will be calling your method with an object that implements the interface, so you don’t need to know the details of the interface; you can just use the object provided to you.

Here’s how it works. You will ask the delegate for its invocation list, and you will call `BeginInvoke` on each delegate in that list. `BeginInvoke` will take two parameters. The first will be a delegate of type `AsyncCallback`, and the second will be your own delegate that invokes the method you want to call:

```
del.BeginInvoke(new AsyncCallback(ResultsReturned),del);
```

In the line of code shown here, you are calling the method encapsulated by `del` (e.g., `DisplayCounter`) and when that method completes, you want to be notified via your method `ResultsReturned`.

The method to be called back (`ResultsReturned`) must match the return type and signature of the `AsyncCallback` delegate: it must return `void` and must take an object of type `IAsyncResult`:

```
private void ResultsReturned(IAsyncResult iar)
{
```

* .NET provides thread pooling, and the “new” thread will typically be pulled from the pool.

When that method is called back, the `IAsyncResult` object is passed in by the .NET Framework. The second parameter to `BeginInvoke` is your delegate, and that delegate is stashed away for you in the `AsyncState` property of the `IAsyncResult` as an `Object`. Inside the `ResultsReturned` callback method, you can extract that `Object` and cast it to its original type:

```
DelegateThatReturnsInt del = (DelegateThatReturnsInt)iar.AsyncState;
```

You can now use that delegate to call the `EndInvoke()` method, passing in the `IAsyncResult` object you received as a parameter:

```
int result = del.EndInvoke(iar);
```

`EndInvoke()` returns the value of the called (and now completed) method, which you assign to a local variable named `result`, and which you are now free to display to the user.

The net effect is that in `Run()`, you get each registered method in turn (first `FirstSubscriber.DisplayCounter` and then `SecondSubscriber.Doubler`), and you invoke each asynchronously. There is no delay between the call to the first and the call to the second, as you aren't waiting for `DisplayCounter` to return.

When `DisplayCounter` (or `Doubler`) has results, your callback method (`ResultsReturned`) is invoked, and you use the `IAsyncResult` object provided as a parameter to get the actual results back from these methods. The complete implementation is shown in Example 12-6.

Example 12-6. Asynchronous invocation of delegates

```
#region Using directives

using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

#endregion

namespace AsynchDelegates
{
    public class ClassWithDelegate
    {
        // a multicast delegate that encapsulates a method
        // that returns an int
        public delegate int DelegateThatReturnsInt();
        public event DelegateThatReturnsInt theDelegate;

        public void Run()
        {
            for ( ; ; )
            {
                // sleep for a half second
            }
        }
    }
}
```

Example 12-6. Asynchronous invocation of delegates (continued)

```
        Thread.Sleep( 500 );

        if ( theDelegate != null )
        {
            // explicitly invoke each delegated method
            foreach (
                DelegateThatReturnsInt del in
                    theDelegate.GetInvocationList() )
            {
                // invoke asynchronously
                // pass the delegate in as a state object
                del.BeginInvoke( new AsyncCallback( ResultsReturned ),
                    del );
            } // end foreach
        } // end if
    } // end for ;;
} // end run

// call back method to capture results
private void ResultsReturned( IAsyncResult iar )
{
    // cast the state object back to the delegate type
    DelegateThatReturnsInt del =
        ( DelegateThatReturnsInt ) iar.AsyncState;

    // call EndInvoke on the delegate to get the results
    int result = del.EndInvoke( iar );

    // display the results
    Console.WriteLine( "Delegate returned result: {0}", result );
}
// end class

public class FirstSubscriber
{
    private int myCounter = 0;

    public void Subscribe( ClassWithDelegate theClassWithDelegate )
    {
        theClassWithDelegate.theDelegate +=
            new ClassWithDelegate.DelegateThatReturnsInt( DisplayCounter );
    }

    public int DisplayCounter()
    {
        Console.WriteLine( "Busy in DisplayCounter..." );
        Thread.Sleep( 10000 );
        Console.WriteLine( "Done with work in DisplayCounter..." );
        return ++myCounter;
    }
}
```

Example 12-6. Asynchronous invocation of delegates (continued)

```
public class SecondSubscriber
{
    private int myCounter = 0;

    public void Subscribe( ClassWithDelegate theClassWithDelegate )
    {
        theClassWithDelegate.theDelegate +=
            new ClassWithDelegate.DelegateThatReturnsInt( Doubler );
    }

    public int Doubler()
    {
        return myCounter += 2;
    }
}

public class Test
{
    public static void Main()
    {
        ClassWithDelegate theClassWithDelegate =
            new ClassWithDelegate();

        FirstSubscriber fs = new FirstSubscriber();
        fs.Subscribe( theClassWithDelegate );

        SecondSubscriber ss = new SecondSubscriber();
        ss.Subscribe( theClassWithDelegate );

        theClassWithDelegate.Run();
    }
}
```