**The C# Station Tutorial**

**by Joe Mayo, 8/25/02, updated 3/12/03, 2/22/08, 1/12/09**

**Lesson 14: Introduction to Delegates and Events**

This lesson introduces *delegates* and *events*. Our objectives are as follows:

* Understand What a *Delegate* Is
* Understand What an *Event* Is
* Implement *Delegates*
* Fire *Events*

**Delegates**

During previous lessons, you learned how to implement reference types using language constructs such as classes and interfaces. These reference types allowed you to create instances of objects and use them in special ways to accomplish your software development goals. Classes allow you to create objects that contained members with attributes or behavior. Interfaces allow you to declare a set of attributes and behavior that all objects implementing them would publicly expose. Today, I'm going to introduce a new reference type called a *delegate*.

A *delegate* is a C# language element that allows you to reference a method. If you were a C or C++ programmer, this would sound familiar because a *delegate* is basically a function pointer. However, developers who have used other languages are probably wondering, "Why do I need a reference to a method?". The answer boils down to giving you maximum flexibility to implement any functionality you want at runtime.

Think about how you use methods right now. You write an algorithm that does its thing by manipulating the values of variables and calling methods directly by name. What if you wanted an algorithm that was very flexible, reusable, and allowed you to implement different functionality as the need arises? Furthermore, let's say that this was an algorithm that supported some type of data structure that you wanted to have sorted, but you also want to enable this data structure to hold different types. If you don't know what the types are, how could you decide an appropriate comparison routine?  Perhaps you could implement an *if/then/else* or *switch* statement to handle well-known types, but this would still be limiting and require overhead to determine the type. Another alternative would be for all the types to implement an interface that declared a common method your algorithm would call, which is actually a nice solution. However, since this lesson is about *delegates*, we'll apply a *delegate* solution, which is quite elegant.

You could solve this problem by passing a *delegate* to your algorithm and letting the contained method, which the *delegate* refers to, perform the comparison operation. Such an operation is performed in Listing 14-1.

**Listing 14-1. Declaring and Implementing a Delegate: SimpleDelegate.cs**

using System;  
  
// this is the delegate declaration  
public delegate int Comparer(object obj1, object obj2);  
  
public class Name  
{  
    public string FirstName = null;  
    public string LastName = null;  
  
    public Name(string first, string last)  
    {  
        FirstName = first;  
        LastName = last;  
    }  
  
    // this is the delegate method handler  
    public static int CompareFirstNames(object name1, object name2)  
    {  
        string n1 = ((Name)name1).FirstName;  
        string n2 = ((Name)name2).FirstName;  
  
        if (String.Compare(n1, n2) > 0)  
        {  
            return 1;  
        }  
        else if (String.Compare(n1, n2) < 0)  
        {  
            return -1;  
        }  
        else  
        {  
            return 0;  
        }  
    }  
  
    public override string ToString()  
    {  
        return FirstName + " " + LastName;  
    }  
}  
  
class SimpleDelegate  
{  
    Name[] names = new Name[5];  
  
    public SimpleDelegate()  
    {  
        names[0] = new Name("Joe", "Mayo");  
        names[1] = new Name("John", "Hancock");  
        names[2] = new Name("Jane", "Doe");  
        names[3] = new Name("John", "Doe");  
        names[4] = new Name("Jack", "Smith");  
    }  
  
    static void Main(string[] args)  
    {  
        SimpleDelegate sd = new SimpleDelegate();  
  
        // this is the delegate instantiation  
        Comparer cmp = new Comparer(Name.CompareFirstNames);  
  
        Console.WriteLine("\nBefore Sort: \n");  
  
        sd.PrintNames();  
  
        // observe the delegate argument  
        sd.Sort(cmp);  
  
        Console.WriteLine("\nAfter Sort: \n");  
  
        sd.PrintNames();  
    }  
  
    // observe  the delegate parameter  
    public void Sort(Comparer compare)  
    {  
        object temp;  
  
        for (int i=0; i < names.Length; i++)  
        {  
            for (int j=i; j < names.Length; j++)  
            {  
                // using delegate "compare" just like  
                // a normal method  
                if ( compare(names[i], names[j]) > 0 )  
                {  
                    temp = names[i];  
                    names[i] = names[j];  
                    names[j] = (Name)temp;  
                }  
            }  
        }  
    }  
  
    public void PrintNames()  
    {  
        Console.WriteLine("Names: \n");  
  
        foreach (Name name in names)  
        {  
            Console.WriteLine(name.ToString());  
        }  
    }  
}

The first thing the program in Listing 14-1 does is declare a *delegate*. *Delegate* declarations look somewhat like methods, except they have the *delegate* modifier, are terminated with a semi-colon (*;*), and have no implementation. Below, is the *delegate* declaration from Listing 14-1.

public delegate int Comparer(object obj1, object obj2);

This *delegate* declaration defines the signature of a delegate handler method that this *delegate* can refer to. The delegate handler method, for the *Comparer* *delegate*, can have any name, but must have a first parameter of type *object*, a second parameter of type *object*, and return an *int* type. The following method from Listing 14-1 shows a delegate handler method that conforms to the signature of the *Comparer* *delegate*.

    public static int CompareFirstNames(object name1, object name2)  
    {  
        ...  
    }

**Note:** The *CompareFirstNames* method calls *String.Compare* to compare the *FirstName* properties of the two *Name* instances. The *String* class has many convenience methods, such as *Compare*, for working with strings. Please don't allow the implementation of this method to interfere with learning how delegates work. What you should concentrate on is that *CompareFirstNames* is a handler method that a delegate can refer to, regardless of the code inside of that method.

To use a *delegate*, you must create an instance of it. The instance is created, similar to a class instance, with a single parameter identifying the appropriate delegate handler method, as shown below.

        Comparer cmp = new Comparer(Name.CompareFirstNames);

The *delegate*, *cmp*, is then used as a parameter to the *Sort()* method, which uses it just like a normal method. Observe the way the *delegate* is passed to the *Sort()* method as a parameter in the code below.

        sd.Sort(cmp);

Using this technique, any delegate handler method may be passed to the *Sort()* method at run-time. i.e. You could define a method handler named *CompareLastNames()*, instantiate a new *Comparer delegate* instance with it, and pass the new *delegate* to the *Sort()* method.

**Events**

Traditional Console applications operate by waiting for a user to press a key or type a command and press the *Enter* key. Then they perform some pre-defined operation and either quit or return to the original prompt that they started from. This works, but is inflexible in that everything is hard-wired and follows a rigid path of execution. In stark contrast, modern GUI programs operate on an event-based model. That is, some event in the system occurs and interested modules are notified so they can react appropriately. With Windows Forms, there is not a polling mechanism taking up resources and you don't have to code a loop that sits waiting for input. It is all built into the system with events.

A C# *event* is a class member that is activated whenever the event it was designed for occurs. I like to use the term "fires" when the *event* is activated. Anyone interested in the *event* can register and be notified as soon as the *event* fires. At the time an *event* fires, registered methods will be invoked.

*Events* and *delegates* work hand-in-hand to provide a program's functionality. It starts with a class that declares an *event*. Any class, including the same class that the *event* is declared in, may register one of its methods for the *event*. This occurs through a *delegate*, which specifies the signature of the method that is registered for the *event*. The *delegate* may be one of the pre-defined .NET *delegates* or one you declare yourself. Whichever is appropriate, you assign the *delegate* to the *event*, which effectively registers the method that will be called when the *event* fires. Listing 14-2 shows a couple different ways to implement *events*.

**Listing 14-2. Declaring and Implementing Events: Eventdemo.cs**

using System;  
using System.Drawing;  
using System.Windows.Forms;  
  
// custom delegate  
public delegate void Startdelegate();  
  
class Eventdemo : Form  
{  
    // custom event  
    public event Startdelegate StartEvent;  
  
    public Eventdemo()  
    {  
        Button clickMe = new Button();  
  
        clickMe.Parent = this;  
        clickMe.Text = "Click Me";  
        clickMe.Location = new Point(  
            (ClientSize.Width - clickMe.Width) /2,  
            (ClientSize.Height - clickMe.Height)/2);  
  
        // an EventHandler delegate is assigned  
        // to the button's Click event  
        clickMe.Click += new EventHandler(OnClickMeClicked);  
  
        // our custom "Startdelegate" delegate is assigned  
        // to our custom "StartEvent" event.  
        StartEvent += new Startdelegate(OnStartEvent);  
  
        // fire our custom event  
        StartEvent();  
    }  
  
    // this method is called when the "clickMe" button is pressed  
    public void OnClickMeClicked(object sender, EventArgs ea)  
    {  
        MessageBox.Show("You Clicked My Button!");  
    }  
  
    // this method is called when the "StartEvent" Event is fired  
    public void OnStartEvent()  
    {  
        MessageBox.Show("I Just Started!");  
    }  
  
    static void Main(string[] args)  
    {  
        Application.Run(new Eventdemo());  
    }  
}

**Note:** If you're using Visual Studio or another IDE, remember to add references to System.Drawing.dll and System.Windows.Forms.dll before compiling Listing 14.2 or just add the code to a Windows Forms project. Teaching the operation of Visual Studio or other IDE's is out-of-scope for this tutorial.

You may have noticed that Listing 14-2 is a Windows Forms program. Although I haven't covered Windows Forms in this tutorial, you should know enough about C# programming in general that you won't be lost. To help out, I'll give a brief explanation of some of the parts that you may not be familiar with.

The *Eventdemo* class inherits *Form*, which essentially makes it a Windows Form. This automatically gives you all the functionality of a Windows Form, including Title Bar, Minimize/Maximize/Close buttons, System Menu, and Borders. A lot of power, that inheritance thing, eh?

The way a Windows Form's application is started is by calling the *Run()* method of the *static* *Application* object with a reference to the *form* object as its parameter. This starts up all the underlying Windows plumbing, displays the GUI, and ensures that *events* are fired as appropriate.

Let's look at the custom *event* first. Below is the *event* declaration, which is a member of the *Eventdemo* class. It is declared with the *event* keyword, a *delegate* type, and an *event* name.

    public event Startdelegate StartEvent;

Anyone interested in an *event* can register by hooking up a *delegate* for that *event*. On the next line, we have a *delegate* of type *Startdelegate*, which the *event* was declared to accept, hooked up to the *StartEvent* *event*. The *+=* syntax registers a *delegate* with an *event*. To unregister with an *event*, use the *-=* with the same syntax.

        StartEvent += new Startdelegate(OnStartEvent);

Firing an *event* looks just like a method call, as shown below:

StartEvent();

This was how to implement *events* from scratch, declaring the *event* and *delegate* yourself. However, much of the *event* programming you'll do will be with pre-defined *events* and *delegates*. This leads us to the other *event* code you see in Listing 14-2, where we hook up an *EventHandler* *delegate* to a *Button* *Click* event.

        clickMe.Click += new EventHandler(OnClickMeClicked);

The *Click* event already belongs to the *Button* class and all we have to do is reference it when registering a *delegate*. Similarly, the *EventHandler* *delegate* already exists in the *System* namespace of the .NET Frameworks Class Library. All you really need to do is define your callback method (delegate handler method) that is invoked when someone presses the *clickMe* button. The *OnClickMeClicked()* method, shown below, conforms to the signature of the *EventHander* *delegate*, which you can look up in the .NET Framework Class Library reference.

    public void OnClickMeClicked(object sender, EventArgs ea)  
    {  
        MessageBox.Show("You Clicked My Button!");  
    }

Any time the *clickMe* button is pressed with a mouse, it will fire the *Click* *event*, which will invoke the *OnClickMeClicked()* method. The *Button* class takes care of firing the *Click* *event* and there's nothing more you have to do. Because it is so easy to use pre-defined *events* and *delegates*, it would be a good idea to check if some exist already that will do what you need, before creating your own.

**Summary**

This completes this lesson, which was an introduction to *delegates* and *events*. You learned how to declare and implement *delegates*, which provide dynamic run-time method invocation services. You also know how to declare *events* and use them in a couple different scenarios. One way is to declare your own *event*, *delegate*, and callback method from scratch. Another way is to use pre-existing *events* and *delegates* and only implement the callback method, which will save you time and make coding easier.

## Tutorial

A delegate in C# is similar to a function pointer in C or C++. Using a delegate allows the programmer to encapsulate a reference to a method inside a delegate object. The delegate object can then be passed to code which can call the referenced method, without having to know at compile time which method will be invoked. Unlike function pointers in C or C++, delegates are object-oriented, type-safe, and secure.

A delegate declaration defines a type that encapsulates a method with a particular set of arguments and return type. For static methods, a delegate object encapsulates the method to be called. For instance methods, a delegate object encapsulates both an instance and a method on the instance. If you have a delegate object and an appropriate set of arguments, you can invoke the delegate with the arguments.

An interesting and useful property of a delegate is that it does not know or care about the class of the object that it references. Any object will do; all that matters is that the method's argument types and return type match the delegate's. This makes delegates perfectly suited for "anonymous" invocation.

**Note**Delegates run under the caller's security permissions, not the declarer's permissions.

This tutorial includes two examples:

* Example 1 shows how to declare, instantiate, and call a delegate.
* Example 2 shows how to combine two delegates.

In addition, it discusses the following topics:

* Delegates and Events
* Delegates vs. Interfaces

#### Example 1

The following example illustrates declaring, instantiating, and using a delegate. The BookDB class encapsulates a bookstore database that maintains a database of books. It exposes a method ProcessPaperbackBooks, which finds all paperback books in the database and calls a delegate for each one. The **delegate** type used is called ProcessBookDelegate. The Test class uses this class to print out the titles and average price of the paperback books.

The use of delegates promotes good separation of functionality between the bookstore database and the client code. The client code has no knowledge of how the books are stored or how the bookstore code finds paperback books. The bookstore code has no knowledge of what processing is done on the paperback books after it finds them.

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode0');" \o "Copy to clipboard.)

// bookstore.cs

using System;

// A set of classes for handling a bookstore:

namespace Bookstore

{

using System.Collections;

// Describes a book in the book list:

public struct Book

{

public string Title; // Title of the book.

public string Author; // Author of the book.

public decimal Price; // Price of the book.

public bool Paperback; // Is it paperback?

public Book(string title, string author, decimal price, bool paperBack)

{

Title = title;

Author = author;

Price = price;

Paperback = paperBack;

}

}

// Declare a delegate type for processing a book:

public delegate void ProcessBookDelegate(Book book);

// Maintains a book database.

public class BookDB

{

// List of all books in the database:

ArrayList list = new ArrayList();

// Add a book to the database:

public void AddBook(string title, string author, decimal price, bool paperBack)

{

list.Add(new Book(title, author, price, paperBack));

}

// Call a passed-in delegate on each paperback book to process it:

public void ProcessPaperbackBooks(ProcessBookDelegate processBook)

{

foreach (Book b in list)

{

if (b.Paperback)

// Calling the delegate:

processBook(b);

}

}

}

}

// Using the Bookstore classes:

namespace BookTestClient

{

using Bookstore;

// Class to total and average prices of books:

class PriceTotaller

{

int countBooks = 0;

decimal priceBooks = 0.0m;

internal void AddBookToTotal(Book book)

{

countBooks += 1;

priceBooks += book.Price;

}

internal decimal AveragePrice()

{

return priceBooks / countBooks;

}

}

// Class to test the book database:

class Test

{

// Print the title of the book.

static void PrintTitle(Book b)

{

Console.WriteLine(" {0}", b.Title);

}

// Execution starts here.

static void Main()

{

BookDB bookDB = new BookDB();

// Initialize the database with some books:

AddBooks(bookDB);

// Print all the titles of paperbacks:

Console.WriteLine("Paperback Book Titles:");

// Create a new delegate object associated with the static

// method Test.PrintTitle:

bookDB.ProcessPaperbackBooks(new ProcessBookDelegate(PrintTitle));

// Get the average price of a paperback by using

// a PriceTotaller object:

PriceTotaller totaller = new PriceTotaller();

// Create a new delegate object associated with the nonstatic

// method AddBookToTotal on the object totaller:

bookDB.ProcessPaperbackBooks(new ProcessBookDelegate(totaller.AddBookToTotal));

Console.WriteLine("Average Paperback Book Price: ${0:#.##}",

totaller.AveragePrice());

}

// Initialize the book database with some test books:

static void AddBooks(BookDB bookDB)

{

bookDB.AddBook("The C Programming Language",

"Brian W. Kernighan and Dennis M. Ritchie", 19.95m, true);

bookDB.AddBook("The Unicode Standard 2.0",

"The Unicode Consortium", 39.95m, true);

bookDB.AddBook("The MS-DOS Encyclopedia",

"Ray Duncan", 129.95m, false);

bookDB.AddBook("Dogbert's Clues for the Clueless",

"Scott Adams", 12.00m, true);

}

}

}

#### Output

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode1');" \o "Copy to clipboard.)

Paperback Book Titles:

The C Programming Language

The Unicode Standard 2.0

Dogbert's Clues for the Clueless

Average Paperback Book Price: $23.97

#### Code Discussion

* **Declaring a delegate**The following statement:

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode2');" \o "Copy to clipboard.)

public delegate void ProcessBookDelegate(Book book);

declares a new delegate type. Each delegate type describes the number and types of the arguments, and the type of the return value of methods that it can encapsulate. Whenever a new set of argument types or return value type is needed, a new delegate type must be declared.

* **Instantiating a delegate**Once a delegate type has been declared, a delegate object must be created and associated with a particular method. Like all other objects, a new delegate object is created with a **new** expression. When creating a delegate, however, the argument passed to the **new** expression is special — it is written like a method call, but without the arguments to the method.

The following statement:

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode3');" \o "Copy to clipboard.)

bookDB.ProcessPaperbackBooks(new ProcessBookDelegate(PrintTitle));

creates a new delegate object associated with the static method Test.PrintTitle. The following statement:

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode4');" \o "Copy to clipboard.)

bookDB.ProcessPaperbackBooks(new

ProcessBookDelegate(totaller.AddBookToTotal));

creates a new delegate object associated with the nonstatic method AddBookToTotal on the object totaller. In both cases, this new delegate object is immediately passed to the ProcessPaperbackBooks method.

Note that once a delegate is created, the method it is associated with never changes — delegate objects are immutable.

* **Calling a delegate**Once a delegate object is created, the delegate object is typically passed to other code that will call the delegate. A delegate object is called by using the name of the delegate object, followed by the parenthesized arguments to be passed to the delegate. An example of a delegate call is:

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode5');" \o "Copy to clipboard.)

processBook(b);

A delegate can either be called synchronously, as in this example, or asynchronously by using [BeginInvoke](http://msdn.microsoft.com/en-us/library/system.windows.forms.control.begininvoke(v=vs.71).aspx) and [EndInvoke](http://msdn.microsoft.com/en-us/library/system.windows.forms.control.endinvoke(v=vs.71).aspx) methods.

#### Example 2

This example demonstrates composing delegates. A useful property of delegate objects is that they can be composed using the "**+**" operator. A composed delegate calls the two delegates it was composed from. Only delegates of the same type can be composed.

The "**-**" operator can be used to remove a component delegate from a composed delegate.

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode6');" \o "Copy to clipboard.)

// compose.cs

using System;

delegate void MyDelegate(string s);

class MyClass

{

public static void Hello(string s)

{

Console.WriteLine(" Hello, {0}!", s);

}

public static void Goodbye(string s)

{

Console.WriteLine(" Goodbye, {0}!", s);

}

public static void Main()

{

MyDelegate a, b, c, d;

// Create the delegate object a that references

// the method Hello:

a = new MyDelegate(Hello);

// Create the delegate object b that references

// the method Goodbye:

b = new MyDelegate(Goodbye);

// The two delegates, a and b, are composed to form c:

c = a + b;

// Remove a from the composed delegate, leaving d,

// which calls only the method Goodbye:

d = c - a;

Console.WriteLine("Invoking delegate a:");

a("A");

Console.WriteLine("Invoking delegate b:");

b("B");

Console.WriteLine("Invoking delegate c:");

c("C");

Console.WriteLine("Invoking delegate d:");

d("D");

}

}

#### Output

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode7');" \o "Copy to clipboard.)

Invoking delegate a:

Hello, A!

Invoking delegate b:

Goodbye, B!

Invoking delegate c:

Hello, C!

Goodbye, C!

Invoking delegate d:

Goodbye, D!

### Delegates and Events

Delegates are ideally suited for use as events — notifications from one component to "listeners" about changes in that component. For more information on the use of delegates for events, see the [Events Tutorial](http://msdn.microsoft.com/en-us/library/aa645739(v=vs.71).aspx).

### Delegates vs. Interfaces

Delegates and interfaces are similar in that they enable the separation of specification and implementation. Multiple independent authors can produce implementations that are compatible with an interface specification. Similarly, a delegate specifies the signature of a method, and authors can write methods that are compatible with the delegate specification. When should you use interfaces, and when should you use delegates?

Delegates are useful when:

* A single method is being called.
* A class may want to have multiple implementations of the method specification.
* It is desirable to allow using a static method to implement the specification.
* An event-like design pattern is desired (for more information, see the [Events Tutorial](http://msdn.microsoft.com/en-us/library/aa645739(v=vs.71).aspx)).
* The caller has no need to know or obtain the object that the method is defined on.
* The provider of the implementation wants to "hand out" the implementation of the specification to only a few select components.
* Easy composition is desired.

Interfaces are useful when:

* The specification defines a set of related methods that will be called.
* A class typically implements the specification only once.
* The caller of the interface wants to cast to or from the interface type to obtain other interfaces or classes.