JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 5, May-June 2004

Delegates and Events in C#

Dr. Richard Wiener, Editor-in-Chief, JOT**,** Associate Professor of Computer Science**,** University of Colorado at Colorado Springs

Delegates are a feature of C# not found in Java.

A delegate is a class that encapsulates a method signature. Although it can be used in any context, it often serves as the basis for the event-handling model in C# but can be used in a context removed from event handling (e.g. passing a method to a method through a delegate parameter).

Delegates provide a type-safe, object-oriented mechanism for treating functions as objects and passing method references as parameters without having to use function pointers as in C or C++.

Since delegates are classes, they have instances (delegate objects). Such delegate instances contain references to one or more methods. A delegate instance is connected to one or more methods using a delegate constructor and later using simple operators. The methods that a delegate instance is associated with (invocation list) are invoked through the delegate instance.

Delegates play a central role throughout the .NET framework and provide the basis for event handling so it is important to acquire a solid understanding of how they work and how they can be used.

A delegate defines the signature (return type and sequence of parameter types) of the methods that can be added to the invocation list of a delegate instance. If an attempt is made to add a method to a delegate object's invocation list that does not confirm precisely to the signature in the delegate class, a compilation error is emitted.

An example of a delegate class declaration is the following:

```
public delegate String SomeDelegate(int x, double y, char z);
```

Instances of a delegate class are created as indicated in the following examples:

```
SomeDelegate d1 = new SomeDelegate(Method1);
SomeDelegate d2 = new SomeDelegate(Method2);
SomeDelegate d3 = d1 + d2;
d3 += new SomeDelegate(Method3);
d3 -= d1;
```

The invocation list for delegate instance *d3* is *Method2* and *Method3*.

An example of invoking the method(s) on the invocation list of a delegate would be the following:

```
String result = d3(16, 4.1, ‘a’);
```

The return value is always associated with the last method in the invocation list, *Method2* in this case. Often delegates are associated with methods that return void (commands) so there is no return value to be concerned about.

Like all classes, delegates can be top-level classes within a namespace or be nested within an existing class. The same access modifiers that can be applied to classes apply for delegates.

Delegates can contain generic parameters or return type.

Listing 1 shows the use of delegates in a generic sorting application. The generic features of the soon to be released Version 1.2 of C# are used in this application.

Listing 1 – A sorting application that shows delegates in action

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

namespace Delegates {

    public delegate void Sorting<T>(ref T [] data, int size)
                      where T : IComparable;

    public class UsingDelegates {

        public UsingDelegates() {
            Sorting<double> s1 =
                                 new Sorting<double>(SelectionSort);
            Sorting<double> s2 = new Sorting<double>(BubbleSort);
            double[] myData = {3.5, 2.1, 5.6, 1.4, 2.5 };
            s1(ref myData, myData.Length);
            double[] sameData = { 3.5, 2.1, 5.6, 1.4, 2.5 };
            s2(ref sameData, sameData.Length);
            Output(myData, myData.Length);
            Output(sameData, sameData.Length);
        }
```

```
public void SelectionSort<T>(ref T [] data, int size)
                  where T : IComparable {
    for (int i = size; i >= 1; i--) {
        // Get the largest value
        T largest = data[0];
        int index = 0;
        for (int j = 1; j < i; j++) {
            if (data[j].CompareTo(largest) > 0) {
                index = j;
                largest = data[j];
            }
        }

        // interchange data[index] and data[i - 1]
        T temp = data[i - 1];
        data[i - 1] = data[index];
        data[index] = temp;
    }
}

public void BubbleSort<T>(ref T[] data, int size) where T:
                              IComparable {
    int index1, index2;
    T temp;
    bool exchanged;
    for (index1 = size - 1; index1 >= 2; index1--) {
        exchanged = false;
        for (index2 = 0; index2 <= index1 - 1; index2++) {
            if (data[index2].CompareTo(data[index2 + 1]) > 0) {
                // Interchange elements at index2 and index2 + 1
                temp = data[index2];
                data[index2] = data[index2 + 1];
                data[index2 + 1] = temp;
                exchanged = true;
            }
        }
        if (!exchanged)
            break;
    }
}

public void Output<T>(T[] data, int size) {
    for (int i = 0; i < size; i++) {
        System.Console.WriteLine(data[i].ToString() + " ");
    }
    System.Console.WriteLine();
}
```

```
        static void Main(string[] args) {
            UsingDelegates app = new UsingDelegates();
        }
    }
}

/* Program output
1.4
2.1
2.5
3.5
5.6

1.4
2.1
2.5
3.5
5.6
*/
```

The delegate class,

```
public delegate void Sorting<T>(ref T [] data, int size)
                           where T : IComparable;
```

establishes a template for any method that might be used as an instance of this delegate. Such a method must be generic, with a constrained generic parameter T that implements *IComparable*. It must have two parameters, the first a reference (in/out semantics) to an array of generic type T and the second parameter, the size of this array.

Two methods are defined that satisfy this template: *SelectionSort* and *BubbleSort*.

Instances of the delegate class *Sorting* are created as follows:

```
Sorting<double> s1 = new Sorting<double>(SelectionSort);
Sorting<double> s2 = new Sorting<double>(BubbleSort);
```

Any generic parameter could be used in place of type *double* providing that its class implements the *IComparable* interface.

Finally, these delegate instances are invoked as follows:

```
double [] myData = {3.5, 2.1, 5.6, 1.4, 2.5 };
s1(ref myData, myData.Length);
double [] sameData = { 3.5, 2.1, 5.6, 1.4, 2.5 };
s2(ref sameData, sameData.Length);
```

Events and delegates

Event types are used in connection with the observer pattern. A collection of registered listeners is notified whenever an event occurs. Objects that are interested in receiving a notification of an event register a delegate instance with the event. An event is always defined with an associated delegate that has been defined and accessible. The **event** keyword is a **delegate** modifier. It must always be used in connection with a delegate.

An event can be triggered only within the class that declared it in contrast to a delegate. When the event is triggered, all delegate instances registered with the event are invoked. An event has the value null if it has no registered listeners.

To provide an illustration of the construction and use of events, we consider the following simple application. In a *Stock* class we model the selling price of a particular stock as a one-dimensional random walk. From a given starting value, the stock goes either up or down (by a random value with equal likelihood from 1 to 4 units on each move). We wish a notification event to be generated each time the value of the stock goes 10 units or more above or 10 units or more below its starting value. A collection of broker objects must receive the event notification from the *Stock* object. We limit the number of changes for a particular stock to 100.

Listing 2 presents a solution to this problem.

Listing 2 – Stocks, brokers and events

```
using System;
using System.Threading;

namespace StockBrokers {

    public delegate void StockNotification(String stockName,
                                int currentValue);

    public class Stock {

        // Fields
        private String name;
        private int initialValue;
        private int currentValue;
        private Random rnd = new Random();
        private int numberChanges;

        // Constructor
        public Stock(String name, int initialValue) {
            this.name = name;
            this.initialValue = initialValue;
            currentValue = initialValue;
        }
        public void Activate() {
            while (numberChanges <= 100) {
                Thread.Sleep(250);
                ChangeStockValue();
            }
        }

        // Event definition
        public event StockNotification stockEvent;

        // Commands
        public void ChangeStockValue() {
            currentValue += rnd.Next(-4, 4);
            numberChanges++;
            if (Math.Abs(currentValue - initialValue) > 10) {
                FireEvent();
            }

        public void FireEvent() {
            stockEvent(name, currentValue);
        }
    }
}


using System;
using System.Collections.Generic;

namespace StockBrokers {

    public class Broker {

        // Fields
        private String brokerName;

        // Constructor
        public Broker(String brokerName, Stock stock) {
            this.brokerName = brokerName;
            stock.stockEvent += new StockNotification(Notify);
            stock.Activate();
        }

        public void Notify(String name, int value) {
            System.Console.WriteLine("Broker name: " +
                    brokerName + " Stock name: " + " value: " + value);
        }
    }
}

using System;
```

```
namespace StockBrokers {

    public class StockApplication {

        static void Main() {
            Stock stock1 = new Stock("Valuable", 160);
            Stock stock2 = new Stock("Less Valuable", 140);
            Stock stock3 = new Stock("Most Valuable", 350);
            Broker b1 = new Broker("Thomas", stock1);
            Broker b2 = new Broker("Evans", stock2);
            Broker b3 = new Broker("Jennings", stock3);

            stock1.Activate();
            stock2.Activate();
            stock3.Activate();
        }

    }
}

/* Program output
Broker name: Thomas Stock name:  value: 171
Broker name: Thomas Stock name:  value: 174
Broker name: Thomas Stock name:  value: 175
Broker name: Thomas Stock name:  value: 177
Broker name: Evans Stock name:  value: 151
Broker name: Evans Stock name:  value: 154
Broker name: Evans Stock name:  value: 155
Broker name: Evans Stock name:  value: 157
Broker name: Jennings Stock name:  value: 361
Broker name: Jennings Stock name:  value: 364
Broker name: Jennings Stock name:  value: 365
Broker name: Jennings Stock name:  value: 367
*/
```

In Listing 2, a top-level delegate class, *StockNotification*, is declared as follows:

```
public delegate void StockNotification(String stockName,
                          int currentValue);
```

In class *Stock*, an event is declared as:

```
public event StockNotification stockEvent;
```

A StockNotification instance, Notify, is created in class Broker and registered with the stockEvent in class *Stock* as follows:

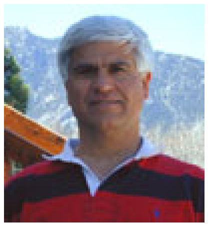
```
stock.stockEvent += new StockNotification(Notify);
```

Whenever the *FireEvent* method is invoked in class Stock's *ChangeStockValue*, the *Notify* method in each Broker object is invoked.

It is clear from the output that each broker object dominates the global stage of the application during the period that its *Notify* method is being sequentially fired by its *Stock* object. To correct this problem we utilize threads.

To be continued …

About the author



Richard Wiener is Associate Professor of Computer Science at the University of Colorado at Colorado Springs. He is also the Editor-in-Chief of JOT and former Editor-in-Chief of the Journal of Object Oriented Programming. In addition to University work, Dr. Wiener has authored or co-authored 21 books and works actively as a consultant and software contractor whenever the possibility arises.