Delegates and Events

People often find it difficult to see the difference between events and delegates. C# doesn't help matters by allowing you to declare *field-like events* which are automatically backed by a delegate variable of the same name. This article aims to clarify the matter for you. Another source of confusion is the overloading of the term "delegate". Sometimes it is used to mean a delegate *type*, and at other times it can be used to mean an *instance* of a delegate type. I'll use "delegate type" and "delegate instance" to distinguish between them, and "delegate" when talking about the whole topic in a general sense.

Delegate types

In some ways, you can think of a delegate type as being a bit like an interface with a single method. It specifies the signature of a method, and when you have a delegate instance, you can make a call to it as if it were a method with the same signature. Delegates provide other features, but the ability to make calls with a particular signature is the reason for the existence of the delegate concept. Delegates hold a reference to a method, and (for instance methods) a reference to the target object the method should be called on.

Delegates types are declared with the `delegate` keyword. They can appear either on their own or nested within a class, as shown below.

```
namespace DelegateArticle
{
    public delegate string FirstDelegate (int x);

    public class Sample
    {
        public delegate void SecondDelegate (char a, char b);
    }
}
```

This code declares two delegate types. The first is `DelegateArticle.FirstDelegate` which has a single parameter of type `int` and returns a `string`. The second is `DelegateArticle.Sample.SecondDelegate` which has two `char` parameters, and doesn't return anything (because the return type is specified as `void`).

Note that the `delegate` keyword doesn't always mean that a delegate type is being declared. The same keyword is used when creating instances of the delegate type using anonymous methods.

The types declared here derive from `System.MulticastDelegate`, which in turn derives from `System.Delegate`. In practice, you'll only see delegate types deriving from `MulticastDelegate`. The difference between `Delegate` and `MulticastDelegate` is largely historical; in betas of .NET 1.0 the difference was significant (and annoying) - Microsoft considered merging the two types together, but decided it was too late in the release cycle to make such a major change. You can pretty much pretend that they're only one type.

Any delegate type you create has the members inherited from its parent types, one constructor with parameters of `object` and `IntPtr` and three extra methods: `Invoke`, `BeginInvoke` and `EndInvoke`. We'll come back to the constructor in a minute. The methods can't be inherited from anything, because the signatures vary according to the signature the delegate is declared with. Using the sample code above, the first delegate has the following methods:

```
public string Invoke (int x);
public System.IAsyncResult BeginInvoke(int x, System.AsyncCallback callback, object state);
public string EndInvoke(IAsyncResult result);
```

As you can see, the return type of `Invoke` and `EndInvoke` matches that of the declaration signature, as are the parameters of `Invoke` and the first parameters of `BeginInvoke`. We'll see the purpose of `Invoke` in the next section, and cover `BeginInvoke` and `EndInvoke` in the section on advanced usage. It's a bit premature to talk about calling methods when we don't know how to create an instance, however. We'll cover that (and more) in the next section.

Delegate instances: the basics

Now we know how a delegate type is declared and what it contains, let's look at how to create an instance of such a type, and what we can do with it.

Creating delegate instances

Note: this article doesn't cover the features of C# 2.0 and 3.0 for creating delegate instances, nor generic delegate variance introduced in C# 4.0. My article on closures talks about the features of C# 2.0 and 3.0 - alternatively, read chapters 5, 9 and 13 of C# in Depth for a lot more detail. By concentrating on the explicit manner of creating instances in C# 1.0/1.1, I believe it will be easier to understand what's going on under the hood. When you understand the basics, it's clearly worth knowing the features these later versions provide - but if you try to use them without having a firm grasp on the basics, you may well get confused.

As mentioned earlier, the key points of data in any particular delegate instance are the method the delegate refers to, and a reference to call the method on (the *target*). For static methods, no target is required. The CLR itself supports other slightly different forms of delegate, where either the first argument passed to a static method is held within the delegate, or the target of an instance method is provided as an argument when the method is called. See the documentation for `System.Delegate` for more information on this if you're interested, but don't worry too much about it.

So, now that we know the two pieces of data required to create an instance (along with the type itself, of course), how do we tell the compiler what they are? We use what the C# specification calls a *delegate-creation-expression* which is of the form `new delegate-type (expression)`. The expression must either be another delegate of the same type (or a compatible delegate type in C# 2.0) or a *method group* - the name of a method and optionally a target, specified as if you were calling the method, but without the arguments or brackets. Creating copies of a delegate is fairly rare, so we will concentrate on the more common form. A few examples are listed below:

```
// The following two creation expressions are equivalent,
// where InstanceMethod is an instance method in the class
// containing the creation expression (or a base class).
// The target is "this".
FirstDelegate d1 = new FirstDelegate(InstanceMethod);
FirstDelegate d2 = new FirstDelegate(this.InstanceMethod);

// Here we create a delegate instance referring to the same method
// as the first two examples, but with a different target.
FirstDelegate d3 = new FirstDelegate(anotherInstance.InstanceMethod);

// This delegate instance uses an instance method in a different class,
// specifying the target to call the method on
FirstDelegate d4 = new FirstDelegate(instanceOfOtherClass.OtherInstanceMethod);

// This delegate instance uses a static method in ths class containing
// the creation expression (or a base class).
FirstDelegate d5 = new FirstDelegate(StaticMethod);

// This delegate instance uses a static method in a different class
FirstDelegate d6 = new FirstDelegate(OtherClass.OtherStaticMethod);
```

The constructor we mentioned earlier has two parameters - an `object` and an `IntPtr`. The `object` is a reference to the target (or `null` for static methods) and the `IntPtr` is a pointer to the method itself.

One point to note is that delegate instances can refer to methods and targets which wouldn't normally be visible at the point the call is actually made. For instance, a private method can be used to create a delegate instance,

and then the delegate instance can be returned from a public member. Alternatively, the target of an instance may be an object which the eventual caller knows nothing about. However, both the target and the method must be accessible to the *creating* code. In other words, if (and only if) you can call a particular method on a particular object, you can use that method and target for delegate creation. Access rights are effectively ignored at call time. Speaking of which...

Calling delegate instances

Delegate instances are called just as if they were the methods themselves. For instance, to call the delegate referred to by variable `d1` above, we could write:

```
string result = d1(10);
```

The method referred to by the delegate instance is called on the target object (if there is one), and the result is returned. Producing a complete program to demonstrate this without including a lot of seemingly irrelevant code is tricky. However, here's a program which gives one example of a static method and one of an instance method. `DelegateTest.StaticMethod` could be written as just `StaticMethod` in the same way that (within an instance method) you could write `InstanceMethod` instead of `this.InstanceMethod` - I've included the class name just to make it clear how you would reference methods from other classes.

```
using System;

public delegate string FirstDelegate (int x);

class DelegateTest
{
    string name;

    static void Main()
    {
        FirstDelegate d1 = new FirstDelegate(DelegateTest.StaticMethod);

        DelegateTest instance = new DelegateTest();
        instance.name = "My instance";
        FirstDelegate d2 = new FirstDelegate(instance.InstanceMethod);

        Console.WriteLine (d1(10)); // Writes out "Static method: 10"
        Console.WriteLine (d2(5));  // Writes out "My instance: 5"
    }

    static string StaticMethod (int i)
    {
        return string.Format ("Static method: {0}", i);
    }

    string InstanceMethod (int i)
    {
        return string.Format ("{0}: {1}", name, i);
    }
}
```

The C# syntax is just a short-hand for calling the `Invoke` method provided by each delegate type. Delegates can also be run asynchronously if they provide `BeginInvoke`/`EndInvoke` methods. These are explained later.

Combining delegates

Delegates can be combined such that when you call the delegate, a whole list of methods are called - potentially with different targets. When I said before that a delegate contained a target and a method, that was a slight simplification. That's what a delegate instance representing one method contains. For the sake of clarity, I'll refer to such delegate instances as *simple delegates*. The alternative is a delegate instance which is effectively a list of simple delegates, all of the same type (i.e. having the same signature). I'll call these *combined delegates*.

Combined delegates can themselves be combined together, effectively creating one big list of simple delegates in the obvious fashion.

It's important to understand that delegate instances are always immutable. Anything which combines them together (or takes one away from the other) creates a new delegate instance to represent the new list of targets/methods to call. This is just like strings: if you call `String.PadLeft` for instance, it doesn't actually change the string you call it on - it just returns a new string with the appropriate padding.

Combining two delegate instances is usually done using the addition operator, as if the delegate instances were strings or numbers. Subtracting one from another is usually done with the subtraction operator. Note that when you subtract one combined delegate from another, the subtraction works in terms of *lists*. If the list to subtract is not found in the original list, the result is just the original list. Otherwise, the *last* occurrence of the list is removed. This is best shown with some examples. Instead of actual code, the following uses lists of simple delegates `d1`, `d2` etc. For instance, `[d1, d2, d3]` is a combined delegate which, when executed, would call `d1` then `d2` then `d3`. An empty list is represented by `null` rather than an actual delegate instance.

Expression	Result
null + d1	d1
d1 + null	d1
d1 + d2	[d1, d2]
d1 + [d2, d3]	[d1, d2, d3]
[d1, d2] + [d2, d3]	[d1, d2, d2, d3]
[d1, d2] - d1	d2
[d1, d2] - d2	d1
[d1, d2, d1] - d1	[d1, d2]
[d1, d2, d3] - [d1, d2]	d3
[d1, d2, d3] - [d2, d1]	[d1, d2, d3]
[d1, d2, d3, d1, d2] - [d1, d2]	[d1, d2, d3]
[d1, d2] - [d1, d2]	null

Delegate instances can also be combined with the static `Delegate.Combine` method, and one can be subtracted from another with the static `Delegate.Remove` method. The C# compiler converts the addition and subtraction operators into calls to these methods. Because they are static methods, they work easily with `null` references.

The addition and subtraction operators always work as part of assignment: `d1 += d2;` is exactly equivalent to `d1 = d1+d2;`, and likewise for subtraction. Again, the original delegate instance remains unchanged; the value of `d1` just changes to be a reference to the appropriate new combined delegate.

Note that because extra delegates are both added to and removed from the end of the list, `x += y; x -= y;` is always a no-op.

If a delegate type is declared to return a value (i.e. it's not declared with a `void` return type) and a combined delegate instance is called, the value returned from that call is the one returned by the last simple delegate in the list.

Events

First things first: events aren't delegate instances. Let's try that again.

Events aren't delegate instances.

It's unfortunate in some ways that C# lets you use them in the same way in certain situations, but it's very important that you understand the difference.

I find the easiest way to understand events is to think of them a bit like properties. While properties *look* like they're fields, they're definitely not - and you can write properties which don't use fields at all. Similarly, while

events look like delegate instances in terms of the way you express the add and remove operations, they're not.

Events are pairs of methods, appropriately decorated in IL to tie them together and let languages know that the methods represent events. The methods correspond to *add* and *remove* operations, each of which take a delegate instance parameter of the same type (the type of the event). What you do with those operations is pretty much up to you, but the *typical* use is to add or remove the delegate from a list of handlers for the event. When the event is triggered (whatever that trigger might be - a button click, a timeout, an unhandled exception) the handlers are called in turn. Note that in C#, the calling of the event handlers is *not* part of the event itself. (CIL defines an association with a `raise_eventName` method, and indeed "other" methods, but these are not used in C#.)

The add and remove methods are called in C# using `eventName += delegateInstance;` and `eventName -= delegateInstance;` respectively, where `eventName` may be qualified with a reference (e.g. `myForm.Click`) or a type name (e.g. `MyClass.SomeEvent`). Static events are relatively rare.

Events themselves can be declared in two ways. The first is with explicit add and remove methods, declared in a very similar way to properties, but with the `event` keyword. Here's an example of an event for the `System.EventHandler` delegate type. Note how it doesn't actually do anything with the delegate instances which are passed to the add and remove methods - it just prints out which operation has been called. Note that the remove operation is called even though we've told it to remove `null`.

```
using System;

class Test
{
    public event EventHandler MyEvent
    {
        add
        {
            Console.WriteLine ("add operation");
        }

        remove
        {
            Console.WriteLine ("remove operation");
        }
    }

    static void Main()
    {
        Test t = new Test();

        t.MyEvent += new EventHandler (t.DoNothing);
        t.MyEvent -= null;
    }

    void DoNothing (object sender, EventArgs e)
    {
    }
}
```

Although it would be very rare to ignore the value in this way, there are times when you don't want to back an event with a simple delegate variable. For instance, in situations where there are lots of events but only a few are likely to be subscribed to, you could have a map from some key describing the event to the delegate currently handling it. This is what Windows Forms does - it means that you can have a huge number of events without wasting a lot of memory with variables which will usually just have `null` values.

A shortcut: field-like events

C# provides a simple way of declaring both a delegate variable and an event at the same time. This is called a *field-like event*, and is declared very simply - it's the same as the "longhand" event declaration, but without the "body" part:

```
public event EventHandler MyEvent;
```

This creates a delegate variable and an event, both with the same type. The access to the event is determined by the event declaration (so the example above creates a public event, for instance) but the delegate variable is always private. The implicit body of the event is the obvious one to add/remove delegate instances to the delegate variable, but the changes are made within a lock. For C# 1.1, the event is equivalent to:

```
private EventHandler _myEvent;

public event EventHandler MyEvent
{
    add
    {
        lock (this)
        {
            _myEvent += value;
        }
    }
    remove
    {
        lock (this)
        {
            _myEvent -= value;
        }
    }
}
```

That's for an instance member. For an event declared as static, the variable is also static and a lock is taken out on `typeof(XXX)` where `XXX` is the name of the class declaring the event. In C# 2.0 there is little guarantee about what is used for locking - only that a single object associated with the instance is used for locking with instance events, and a single object associated with the class is used for locking static events. (Note that this only holds for *class* events, not *struct* events - there are issues in terms of locking with struct events; in practice I don't remember ever seeing a struct with any events.) None of this is actually as useful as you might think - see the threading section for more details.

So, what happens when you refer to `MyEvent` in code? Well, within the text of type itself (including nested types) the compiler generates code which refers to the delegate variable (`_myEvent` in my sample above). In all other contexts, the compiler generates code which refers to the event.

What's the point?

Now we know what they are, what's the point of having both delegates and events? The answer is encapsulation. Suppose events didn't exist as a concept in C#/.NET. How would another class subscribe to an event? Three options:

1. A public delegate variable
2. A delegate variable backed by a property
3. A delegate variable with `AddXXXHandler` and `RemoveXXXHandler` methods

Option 1 is clearly horrible, for all the normal reasons we abhor public variables. Option 2 is slightly better, but allows subscribers to effectively override each other - it would be all too easy to write `someInstance.MyEvent = eventHandler;` which would *replace* any existing event handlers rather than adding a new one. In addition, you still need to write the properties.

Option 3 is basically what events give you, but with a guaranteed convention (generated by the compiler and backed by extra flags in the IL) and a "free" implementation if you're happy with the semantics that field-like events give you. Subscribing to and unsubscribing from events is encapsulated without allowing arbitrary access to the list of event handlers, and languages can make things simpler by providing syntax for both declaration and subscription.

Thread-safe events

(Note: this section needs a certain amount of revision in the light of C# 4.)

Earlier we touched on field-like events locking during the add/remove operations. This is to provide a certain amount of thread safety. Unfortunately, it's not terribly useful. Firstly, even with 2.0, the spec allows for the lock to be the reference to `this` object, or the type itself for static events. That goes against the principle of locking on privately held references to avoid accidental deadlocks.

Ironically, the second problem is the exact reverse of the first - because in C# 2.0 you can't guarantee which lock is going to be used, you can't use it yourself when raising an event to ensure that you see the most recent value in the thread doing the raising. You can lock on something else or call one of the memory barrier methods, but it leaves something of a nasty taste in the mouth.

If you want to be truly thread-safe, such that when you raise an event you always use the most recent value of the delegate variable, along with making sure that the add/remove operations don't interfere with each other, you need to write the body of the add/remove operations yourself. Here's an example:

```
/// <summary>
/// Delegate variable backing the SomeEvent event.
/// </summary>
SomeEventHandler someEvent;

/// <summary>
/// Lock for SomeEvent delegate access.
/// </summary>
readonly object someEventLock = new object();

/// <summary>
/// Description for the event
/// </summary>
public event SomeEventHandler SomeEvent
{
    add
    {
        lock (someEventLock)
        {
            someEvent += value;
        }
    }
    remove
    {
        lock (someEventLock)
        {
            someEvent -= value;
        }
    }
}

/// <summary>
/// Raises the SomeEvent event
/// </summary>
protected virtual void OnSomeEvent(EventArgs e)
{
    SomeEventHandler handler;
    lock (someEventLock)
    {
        handler = someEvent;
    }
    if (handler != null)
    {
        handler (this, e);
    }
}
```

You could use a single lock for all your events, or even for other things as well - it depends on your situation. Note that you need to assign the current value to a local variable inside the lock (to get the most recent value) and then test it for nullity and execute it *outside* the lock: holding the lock whilst raising the event is a very bad idea, as you could easily deadlock. (Event handlers may well need to wait for another thread to do something, and if that other thread were to call the add or remove operation on your event, you'd get deadlock.)

This all works because once `handler` has been assigned the value of `someEvent`, the value of `handler` won't change even if `someEvent` does. So if all the handlers are unsubscribed from the event, `someEvent` will become `null` but `handler` will still have whatever value it had when it was assigned. In fact, as delegate instances are immutable, whatever handlers were subscribed when the `handler = someEvent;` line was executed will be called, even if others have subscribed between then and the `handler (this, e);` line.

Now, it's important to consider whether or not you even *need* thread safety. Are event handlers going to be added or removed from other threads? Do you need to raise the event from another thread? If you're in complete control of your application, the answer may very well be "no". (If you're writing a class library, it's more likely that being thread-safe is important.) If you don't need thread safety, you may want to implement the add/remove operations to get round the problem of the externally visible lock that C# uses (or may use in the case of 2.0). At that point, the operations become pretty trivial. Here's the equivalent of the earlier code, but without thread safety.

```
/// <summary>
/// Delegate variable backing the SomeEvent event.
/// </summary>
SomeEventHandler someEvent;

/// <summary>
/// Description for the event
/// </summary>
public event SomeEventHandler SomeEvent
{
    add
    {
        someEvent += value;
    }
    remove
    {
        someEvent -= value;
    }
}

/// <summary>
/// Raises the SomeEvent event
/// </summary>
protected virtual void OnSomeEvent(EventArgs e)
{
    if (someEvent != null)
    {
        someEvent (this, e);
    }
}
```

The check for nullity is due to delegate variables being `null` when there aren't any delegate instances to call. One way to make things simpler is to use a no-op delegate instance as the "default" one, which is never removed. At that point, you can just obtain the value of the delegate variable (inside a lock if you're being thread-safe) and then execute the delegate instance. If there are no "real" delegate targets to call, the no-op target will execute and that's all that will happen.

Delegate instances: other methods

Earlier we saw how a call to `someDelegate(10)` is actually a short-hand for `someDelegate.Invoke(10)`. Delegates types may also allow asynchronous behaviour using the `BeginInvoke`/`EndInvoke` pair. These are optional as far as the CLI specification is concerned, but C# delegate types always provide them. They follow the same model for

asynchronous execution as the rest of .NET, allowing a callback handler to be provided, along with an object to store state information. The delegates are executed on threads created by the system thread-pool.

The first example below operates without a callback, simply using `BeginInvoke` and `EndInvoke` from the same thread. This is occasionally useful when a single thread is used for an operation which is synchronous in general, but which contains elements which may be performed in parallel. The methods involved are all static for the sake of simplicity, but delegate instances with specific target objects can also be used, and often are. `EndInvoke` returns whatever value was returned by the delegate call. If the call threw an exception, the same exception is thrown by `EndInvoke`.

```
using System;
using System.Threading;

delegate int SampleDelegate(string data);

class AsyncDelegateExample1
{
    static void Main()
    {
        SampleDelegate counter = new SampleDelegate(CountCharacters);
        SampleDelegate parser = new SampleDelegate(Parse);

        IAsyncResult counterResult = counter.BeginInvoke ("hello", null, null);
        IAsyncResult parserResult = parser.BeginInvoke ("10", null, null);
        Console.WriteLine ("Main thread continuing");

        Console.WriteLine ("Counter returned {0}", counter.EndInvoke(counterResult));
        Console.WriteLine ("Parser returned {0}", parser.EndInvoke(parserResult));

        Console.WriteLine ("Done");
    }

    static int CountCharacters (string text)
    {
        Thread.Sleep (2000);
        Console.WriteLine ("Counting characters in {0}", text);
        return text.Length;
    }

    static int Parse (string text)
    {
        Thread.Sleep (100);
        Console.WriteLine ("Parsing text {0}", text);
        return int.Parse(text);
    }
}
```

The calls to `Thread.Sleep` are just to demonstrate that the execution really does occur in parallel. The sleep in `CountCharacters` is as large as it is to force the system thread-pool to run the tasks on two different threads - the thread-pool serializes requests which don't take long in order to avoid creating more threads than it needs to. By sleeping for a long time, we're simulating a long-running request. Here's the output from a sample run:

```
Main thread continuing
Parsing text 10
Counting characters in hello
Counter returned 5
Parser returned 10
Done
```

The calls to `EndInvoke` block until the delegate has completed in much the same way as calls to `Thread.Join` block until the threads involved have terminated. The `IAsyncResult` values returned by the calls to `BeginInvoke` allows access to the state passed as the last parameter to `BeginInvoke`, but this isn't typically used in the style of asynchronous invocation shown above.

The code above is fairly simple, but often not as powerful as a model which uses callbacks after the delegate has completed. Typically, the callback will call `EndInvoke` to obtain the result of the delegate. Although it is still a theoretically blocking call, it will never actually block because the callback only executes when the delegate has completed anyway. The callback may well use the state provided to `BeginInvoke` as extra context information. The sample code below uses the same counting and parsing delegates as the previous example, but with a callback displaying the results. The state is used to determine how to format each result, so a single callback can be used for both asynchronous calls. Note the cast from `IAsyncResult` to `AsyncResult`: the value provided to the callback is always an instance of `AsyncResult`, and this can be used to obtain the original delegate instance, so that the callback can call `EndInvoke`. It is somewhat anomalous that `AsyncResult` lives in the `System.Runtime.Remoting.Messaging` namespace when all the other classes involved are in either `System` or `System.Threading`, but such is life.

```
using System;
using System.Threading;
using System.Runtime.Remoting.Messaging;

delegate int SampleDelegate(string data);

class AsyncDelegateExample2
{
    static void Main()
    {
        SampleDelegate counter = new SampleDelegate(CountCharacters);
        SampleDelegate parser = new SampleDelegate(Parse);

        AsyncCallback callback = new AsyncCallback (DisplayResult);

        counter.BeginInvoke ("hello", callback, "Counter returned {0}");
        parser.BeginInvoke ("10", callback, "Parser returned {0}");

        Console.WriteLine ("Main thread continuing");

        Thread.Sleep (3000);
        Console.WriteLine ("Done");
    }

    static void DisplayResult(IAsyncResult result)
    {
        string format = (string) result.AsyncState;
        AsyncResult delegateResult = (AsyncResult) result;
        SampleDelegate delegateInstance = (SampleDelegate)delegateResult.AsyncDelegate;

        Console.WriteLine (format, delegateInstance.EndInvoke(result));
    }

    static int CountCharacters (string text)
    {
        Thread.Sleep (2000);
        Console.WriteLine ("Counting characters in {0}", text);
        return text.Length;
    }

    static int Parse (string text)
    {
        Thread.Sleep (100);
        Console.WriteLine ("Parsing text {0}", text);
        return int.Parse(text);
    }
}
```

This time almost all the work is done on thread-pool threads. The main thread just kicks off the asynchronous tasks and then sleeps for long enough to let all the work finish. (Thread-pool threads are background threads - without the extra `Sleep` call, the application would terminate before the delegate calls finished executing.) Some sample output is below - notice how this time, because there is no guaranteed ordering to the calls to `EndInvoke`, the parser result is displayed before the counter result. In the previous example, the parser almost certainly completed before the counter did, but the main thread waited to obtain the result of the counter first.

```
Main thread continuing
Parsing text 10
Parser returned 10
Counting characters in hello
Counter returned 5
Done
```

Note that you *must* call `EndInvoke` when you use asynchronous execution in order to guarantee not to leak memory or handles. Some implementations may not leak, but you shouldn't rely on this. See my thread-pool article for some sample code to allow "fire and forget" style asynchronous behaviour if this is inconvenient.

Conclusion

Delegates provide a simple way of representing a method call, potentially with a target object, as a piece of data which can be passed around. They are the basis for events, which are effectively conventions for adding and removing handler code to be invoked at appropriate times.