**Fun with Actions and Funcs**

Last updated Dec 25, 2009. <http://www.informit.com/guides/content.aspx?g=dotnet&seqNum=778&ns=16409>

The Action and Func generic delegates were introduced with the other support of delegates in .NET 2.0. These generic delegate types greatly simplified the sometimes very cumbersome process of declaring, defining, and using delegates.

The delegates come in many forms. [Action](http://msdn.microsoft.com/en-us/library/system.action.aspx), for example, is a delegate for a method that takes no parameters and returns no value. [Action<T>](http://msdn.microsoft.com/en-us/library/018hxwa8.aspx) takes one parameter and returns no value. [Func<T, TResult>](http://msdn.microsoft.com/en-us/library/bb549151.aspx) takes no parameters and returns a value. There are defined generic Action and Func delegates that take up to four parameters.

If you're familiar with defining your own delegates, then you'll find Action and Func to be much, much easier. If you're not yet familiar with defining your own delegates, I highly recommend that you use these generic delegate types instead of the old syntax unless you find that you need to do something that these won't do.

This article isn't a primer on using Action and Func, but rather an exploration of a couple things that those delegate types make easier.

**Simplifying that Invoke call**

If you've written any Windows Forms code that handles asynchronous events from other threads, you're familiar with the use of InvokeRequired and Invoke. I covered those briefly in my article about [Asynchronous Callbacks](http://www.informit.com/guides/content.aspx?g=dotnet&seqNum=194).

Before the advent of the Action delegate type, typical code to properly handle the possibility of being called on a different thread (a thread other than the UI thread) would look something like this:

*[C#]*

delegate void SetLabelTextDelegate(string txt);

private void SetLabelText(string txt)

{

if (this.InvokeRequired)

Invoke(new SetLabelTextDelegate(SetLabelText), txt);

else

label1.Text = txt;

}

*[Visual Basic]*

Delegate Sub SetLabelTextDelegate(ByVal txt As String)

Private Sub SetLabelText(ByVal txt As String)

If Me.InvokeRequired Then

Invoke(New SetLabelTextDelegate(AddressOf SetLabelText), txt)

Else

Label1.Text = txt

End If

End Sub

Having to define that delegate type is just annoying, and it becomes tiresome when you have dozens of methods that require those Invoke calls. You can simplify things a bit by using the same delegate type for all methods that have the same types of parameters, so if you had methods SetLabel1Text and SetLabel2Text, you could use the same delegate type for both. But it's still annoying, and I found it difficult to work with.

Action lets you forego defining your own delegate type, simplifying the code.

*[C#]*

private void SetLabelText(string txt)

{

if (this.InvokeRequired)

Invoke(new Action<string>(SetLabelText), txt);

else

label1.Text = txt;

}

*[Visual Basic]*

Private Sub SetLabelText(ByVal txt As String)

If InvokeRequired Then

Me.Invoke(New Action(Of String)(AddressOf SetLabelText), txt)

Else

Label1.Text = txt

End If

End Sub

This works because Action inherits from System.Delegate, and Invoke requires a delegate type. I find this code to be much easier to write because I don't have to pre-define a delegate type. It's a bit more difficult to read at first, but I found that once I had adopted this pattern and worked with it for a while it was quite readable.

Since C# supports anonymous methods, you could also write this:

Invoke(new Action(() => { label1.Text = txt; }));

I would recommend against that, though, as it's not particularly good practice to duplicate code.

**A recursive Func**

A while back I was faced with the need to create a Func that called itself, and found myself stumped for a few minutes with how to do it. Fortunately I was working in C#, because what I want to do doesn't appear to be possible at all in Visual Basic.

The details of the particular function aren't terribly important to this discussion, so I'll just use that old standby recursive function, factorial that we all learned about in CS 101. The standard function definition in C# would look like this.

*[C#]*

static int Factorial(int n)

{

if (n == 1)

return n;

return n \* Factorial(n - 1);

}

But suppose we want that function to be defined within a method. For example:

*[C#]*

void Something()

{

var Fact = new Func<int, int>((n) => { if (n == 1) return n; return n \* Fact(n - 1); });

}

In C#, the compiler issues an error message: "Cannot use local variable 'Fact' before it is declared." However, it doesn't say that recursion isn't possible. What would happen, I wondered, if I declared the variable first?

*[C#]*

Func<int, int> Fact;

Fact = new Func<int, int>((n) => { if (n == 1) return n; return n \* Fact(n - 1); });

The error message changes to "Use of unassigned local variable 'Fact'."

That makes sense. The compiler has to parse the Lambda expression before it can generate code for the assignment, and since the function calls Fact, which hasn't yet been assigned a value, the compiler issues an error message. So what happens if I just go ahead and assign a value?

*[C#]*

Func<int, int> Fact = null;

Fact = new Func<int, int>((n) => { if (n == 1) return n; return n \* Fact(n - 1); });

Eureka! To define a recursive function with a Lambda expression, first declare the variable and assign it a value of null. Then write an assignment statement that includes the Lambda expression.

On the face of it, this looks like it belongs in a "stupid programmer tricks" blog or something, but it turns out to be quite useful in some situations. Because Lambda expressions have access to the local variables in the method in which it's defined, I was saved the work of creating a new method to which I'd have to pass a whole bunch of local variables.