Quick Byte: Get Func<>-y

A quick intro to Func<T, TResult> by JeremyBytes.com

Overview

If you look into the LINQ extension methods, you will run across Func<T, TResult> quite a bit. If you see a Func<> in a method definition, you can treat it like a big sign that says "Put your lambda expression here." What we'll see is that Func<> is simply shorthand for creating a delegate. So, let's take a look at a simple delegate example. Note: this sample is very similar to the sample that Microsoft provides in the Visual Studio Help for Func<T, TResult>.

Creating an Explicit Delegate

We'll start out with an explicit delegate that we have already defined. You can download the sample code here: http://www.jeremybytes.com/Downloads.aspx. The UI contains an input text box, a button, and an output text block. Here's the code behind the form (from MainWindow.xaml.cs):

```
using System;
using System.Windows;

namespace GetFuncY
{
    delegate string ConvertText(string input);

    public partial class MainWindow : Window
    {
        public MainWindow() ... // collapsed for clarity

        private void ProcessButton_Click(object sender, RoutedEventArgs e)
        {
            ConvertText convert = ConvertToUpper;
            OutputBox.Text = convert(InputBox.Text);
        }

        private string ConvertToUpper(string input)
        {
            return input.ToUpper();
        }
    }
}
```

First, notice that we have an explicit delegate (`ConvertText`) that takes a string as a parameter and returns a string as the result. Next, notice that we have a private method (`ConvertToUpper`) that matches that signature. This method simply converts the input string to uppercase and returns the result.

Now let's look at the button click event handler. Inside the event handler, we have created an instance of our delegate (`convert`) and assigned our private method to it. Then we invoke the delegate using the input text box contents as the parameter and assigning the result to the output text block. The running application looks like this:

This is obviously a very trivial example of setting up a delegate, and in the real world we probably wouldn't use a delegate like this. But we want to keep things simple to follow so we can concentrate on the Func<>-y parts.

Getting Func<>-Y

If we look at the Visual Studio Help for Func<T, TResult>, we have the following definition: "Encapsulates a method that has one parameter and returns a value of the type specified by the *TResult* parameter." And the syntax is defined as follows:

```
public delegate TResult Func<in T, out TResult>(
    T arg
)
```

What this means is that Func<T, TResult> is a delegate that expects a T as the parameter, and TResult as the return type. There are also other versions of Func, such as Func<T1, T2, TResult> which takes 2 parameters and returns a result. In .NET 3.5, this goes up to a total of 4 parameters (T1 – T4); .NET 4.0 has additional methods that go all the way up to 16 parameters (T1 – T16).

You'll note that in our example above, our delegate takes a string parameter and has a string return type. This means that instead of explicitly defining our `ConvertText` delegate, we can simply replace it with a `Func<string, string>`.

```
using System;
using System.Windows;

namespace GetFuncY
{
    public partial class MainWindow : Window
    {
        public MainWindow() ... // collapsed for clarity
```

```
        private void ProcessButton_Click(object sender, RoutedEventArgs e)
        {
            Func<string, string> convert = ConvertToUpper;
            OutputBox.Text = convert(InputBox.Text);
        }

        private string ConvertToUpper(string input)
        {
            return input.ToUpper();
        }
    }
}
```

Notice that our `delegate` declaration at the top of our namespace is gone. And instead of the `convert` variable being of type `ConvertText`, it is now of type `Func<string, string>`. This still means the same thing, and our `ConvertToUpper` method still matches the delegate signature (a single string parameter and a string return type). The only difference is that we do not have to explicitly define a separate delegate. We can simply use Func<>. If we run the application now, we will get exactly the same results.

Moving to an Anonymous Delegate

We can do a little more refactoring by converting our named delegate to an anonymous delegate. We do this by in-lining the body of the `ConvertToUpper` method. Here's what that looks like:

```
using System;
using System.Windows;

namespace GetFuncY
{
    public partial class MainWindow : Window
    {
        public MainWindow() ...// collapsed for clarity

        private void ProcessButton_Click(object sender, RoutedEventArgs e)
        {
            Func<string, string> convert = delegate(string input)
                { return input.ToUpper(); };
            OutputBox.Text = convert(InputBox.Text);
        }
    }
}
```

In the assignment to `convert`, we use the `delegate` keyword, and then include the parameter and body from the `ConvertToUpper` method. Since all of that code is in-line, we can remove the `ConvertToUpper` method. This delegate is now known as an anonymous delegate since it no longer has an explicit name (although the compiler will give it a name internally).

Converting to a Lambda

A lambda expression is simply an anonymous delegate (with a few added features). This means that we can further replace our anonymous delegate with a lambda expression by just removing the `delegate` keyword and adding the "goes to" operator (=>) between the parameters and the method body:

```
using System;
using System.Windows;

namespace GetFuncY
{
    public partial class MainWindow : Window
    {
        public MainWindow() ...// collapsed for clarity

        private void ProcessButton_Click(object sender, RoutedEventArgs e)
        {
            Func<string, string> convert = (string input) =>
                { return input.ToUpper(); };
            OutputBox.Text = convert(InputBox.Text);
        }
    }
}
```

And since lambda expressions allow us to remove the parameter type, the parentheses (since we only have one parameter), and the curly braces (since we only have one statement), we can reduce this even further:

```
using System;
using System.Windows;

namespace GetFuncY
{
    public partial class MainWindow : Window
    {
        public MainWindow() ...// collapsed for clarity

        private void ProcessButton_Click(object sender, RoutedEventArgs e)
        {
            Func<string, string> convert = input => input.ToUpper();
            OutputBox.Text = convert(InputBox.Text);
        }
    }
}
```

If we run the application, we will see that we have exactly the same behavior.

Relating Func<> and Lambdas

What we can see is that Func<T, TResult> and lambda expressions go very well together. The "T" becomes our parameter for the lambda. (If we had a Func<T1, T2, TResult>, then both the "T1" and "T2" would be parameters for the lambda.) And the "TResult" is the type that we return from the lambda body.

This means that if we are using LINQ and find that the Enumerable.Where has a Func<TSource, bool> as a parameter, we can replace this with a lambda expression that has TSource as the parameter and returns a true/false value. If you want to see more of how lambdas and LINQ work together, you can download "Learn to Love Lambdas" from the website: http://www.jeremybytes.com/Downloads.aspx.

Wrap Up

Func<T, TResult> is simply a way of specifying the signature of a delegate. This can save us from having to add an explicit declaration in our code. But more importantly, if a method is expecting a Func<> as a parameter, then we can simply drop in a lambda expression. And this is the best way of handling things when dealing with LINQ extension methods. Hopefully you have a better understanding of Func<> and how it can make your code more readable and lambda-friendly.

Happy Coding!