Quick Byte: Statement Lambdas

A quick intro to Statement Lambdas by JeremyBytes.com

Overview

The first time I came across a lambda expression, I was perplexed. I could tell that something important was going on, but I got stuck on the new syntax:

```
proxy.GetPeopleCompleted +=
    (serviceSender, serviceArgs) => PersonListBox.ItemsSource = serviceArgs.Result;
```

Let's break it down to make things a little easier. A lambda expression consists of 3 parts:

1. Parameters – these appear on the left side of the operator.
2. => – this is the "goes to" operator that denotes a lambda expression.
3. Expressions or Statements – these appear on the right side of the operator.

Expression Lambdas evaluate to a true/false value. You often see these in LINQ and extension methods. Statement Lambdas consist of an action (or set of actions). We're looking at Statement Lambdas here.

The short version is that a statement lambda is simply an anonymous delegate. Let's go step by step to see how we can take a standard delegate (event handler) and turn it into a statement lambda.

A Standard Event Handler

Our scenario is a quick Silverlight application (you can download the code here if you'd like to see for yourself: http://www.jeremybytes.com/Downloads.aspx). This consists of a PersonService that has a GetPeople() method that returns a list of names. In our Silverlight UI, we have a ListBox that we want to populate with the names when we click a button.

Using a standard event handler, the implementation code looks like this:

```
private void LoadDataButton_Click(object sender, RoutedEventArgs e)
{
   var proxy = new PersonServiceClient();
   proxy.GetPeopleCompleted +=
      new System.EventHandler<GetPeopleCompletedEventArgs>(proxy_GetPeopleCompleted);
   proxy.GetPeopleAsync();
}

void proxy_GetPeopleCompleted(object sender, GetPeopleCompletedEventArgs e)
{
    PersonListBox.ItemsSource = e.Result;
}
```

In the Button Click event, the first step is to create a proxy to our WCF service. Then we hook up the GetPeopleCompleted call back handler. This is where we put our code to handle the data that comes back from our service. The last step is to call the GetPeopleAsync() method to kick off the data fetch.

To create the event handler, we let Visual Studio do the work for us. If we type "proxy.GetPeopleCompleted +=", then IntelliSense will offer to create a new event handler for us. If we press Tab, it will offer a name for the event handler. If we press Tab again, it will create the implementation stub for us.

To finish things off we implement the body of the event handler. In this case, we simply get the result of our service and assign it to the ItemsSource of the ListBox.

Creating an Anonymous Delegate

An event handler is simply a delegate. In the example we just created, it is a named delegate. But C# also gives us the option of creating an anonymous delegate. This, as you may guess, is a delegate without a name.

Here's what our modified code looks like:

```
private void LoadDataButton_Click(object sender, RoutedEventArgs e)
{
    var proxy = new PersonServiceClient();
    proxy.GetPeopleCompleted +=
        delegate(object serviceSender, GetPeopleCompletedEventArgs serviceArgs)
        {
            PersonListBox.ItemsSource = serviceArgs.Result;
        };
    proxy.GetPeopleAsync();
}
```

To create the anonymous delegate, we simply in-line our event handler. You'll notice that the parameters (`object` and `GetPeopleCompletedEventArgs`) are the same, and the body of the method is the same. The new part is that we added the `delegate` keyword before the parameters. We also renamed our delegate parameters because we already have a "sender" and "e" within the current scope – so we renamed them to "serviceSender" and "serviceArgs".

Running the application now will give us the same result as with the named event handler.

Creating a Lambda Expression

The next step is to turn this into a lambda expression. Here's what our modified code looks like:

```
private void LoadDataButton_Click(object sender, RoutedEventArgs e)
{
    var proxy = new PersonServiceClient();
    proxy.GetPeopleCompleted +=
        (object serviceSender, GetPeopleCompletedEventArgs serviceArgs) =>
```

```
                {
                    PersonListBox.ItemsSource = serviceArgs.Result;
                };
            proxy.GetPeopleAsync();
        }
```

You can see that the only difference here is that we removed the `delegate` keyword and added the "goes to" operator (=>) to the right of the parameters. Now we have a lambda expression

Shortening the Lambda Expression

Lambda expression syntax has a few features that let us shorten our statement a bit more. First, if the compiler can determine the types for the parameters, you do not need to explicitly state those types.

```
    private void LoadDataButton_Click(object sender, RoutedEventArgs e)
    {
        var proxy = new PersonServiceClient();
        proxy.GetPeopleCompleted +=
            (serviceSender, serviceArgs) =>
            {
                PersonListBox.ItemsSource = serviceArgs.Result;
            };
        proxy.GetPeopleAsync();
    }
```

If you hover the cursor over `serviceArgs` you will see that it is still strongly-typed (`GetPeopleCompletedEventArgs`), but you do not need to specify it since the compiler can determine the type.

Finally, since we only have a single statement in our method body, we can remove the curly braces. This leaves us with the following:

```
private void LoadDataButton_Click(object sender, RoutedEventArgs e)
{
  var proxy = new PersonServiceClient();
  proxy.GetPeopleCompleted +=
    (serviceSender, serviceArgs) => PersonListBox.ItemsSource = serviceArgs.Result;
  proxy.GetPeopleAsync();
}
```

And this is exactly where we started.

So you can see that statement lambda expressions are simply anonymous delegates, and anonymous delegates are simply in-lined named delegates. This was my "aha" moment with statement lambdas. Hopefully it will help you out, too.

Happy Coding!