Lab 6: Lambdas

Programming in C# for Visual Studio .NET Platform

Objective

When delegates are used for callbacks as shown in prior exercises, it is quite common for the callback method to have very little code in it. It can also be cumbersome in a large application to have a lot of callback methods that are referenced in a single location, but need to be defined in a separate method.

In this lab exercise, we look at various ways that we can write code that provides the body of a simple method "in-line" at the single point where the method is needed.

Procedure

Step 1: For this exercise, you can unzip the LambdasLab.zip file from Canvas to your course directory as a starter solution. Take a moment to look at the organization of the solution. You will note that there are three different Console application projects and two Library projects, and each of the Console projects has references to the supporting library projects it needs to use.

Step 2: An awkward issue with callback delegates. I have included Delegate Example #3 from the Delegates Lab in this new solution, so we can take a closer look at it. You can make the DelegateEx03Console project the start-up project for your solution and run it to review this example. It uses a few classes that are found in the DelegateLibrary project, so it needs a project reference to the DelegateLibrary project.

The example does works, but you will notice that where we declare and instantiate the delegate like this:

```
Transformer<double> squareDelegate = Square;
```

The right side of the assignment expects either an explicit call to a delegate constructor or a method name (as in this case), so we need to have a method somewhere named Square with the appropriate signature and return type. There are a few places where we could implement the Square method:

- If we think it is a method that is generally useful in many different contexts, we might define some sort of static Utility class that serves as a library of useful methods. Depending on how widely used we expect the method to be, the Utility class might be in a separate library assembly.
- If the method is very simple or specific to the context of the class we are working in, it might make more sense to make it a static or instance method in the current class as we did in this example.

When working with delegates, you will frequently find that the methods you are creating are very simple – to the point where creating any kind of method seems like overkill. Our example is typical where we just want to square a number, but had to create a completely separate method for this operation.

If you are reading this code and need to check what the Square method does, you need to jump to some other part of the code where the method is implemented just to confirm that the Square method just takes the Square of the input parameter. This isn't a big deal for a simple example like this, but imagine that you are working with a class containing hundreds of lines of code, and dozens of similar simple methods. Even if the methods are just defined right in the class itself as in this example, how would you want to organize them? Should the methods be grouped together in a special code region for one-off methods, or maybe each such method should appear right after the method that uses it? In the subsequent labs we are going to start using a lot of callback delegates, so this practice of having scads of trivial methods scattered about would quickly become quite awkward.

Not only that, but to accommodate the three different data types in the example, we needed to create three different overloads of the Square method that differ only in the data type of the variables. (By the way, you might think that we could create a generic Square method, but this doesn't work, since there is no way to specify a type constraint that assures the type T supports the * operator. See the section of the Generics chapter in Michaelis "Operator Constraints Are Not Allowed".)

The focus of this entire lab exercise is to look at ways to define methods that are simple and used only in one place "in-line" at the point where they are used.

Step 3: Anonymous Methods. In Version 2.0 of the .Net Framework, C# added a feature called ***anonymous methods***, which allow you to insert the code for a method "in-line" at the place where you use it. This effectively creates a method that doesn't even have a name for one-time use within the context of the place where it is used.

First we will create a copy of the DelegateEx03Console project called AnonymousMethodConsole, which we can then modify to use anonymous methods:

1. Create a new Console project in the solution and name it AnonymousMethodConsole.
2. Add to this project a reference to the DelegateLibrary project.
3. Delete the Program.cs file from the new console project.
4. Copy and paste the Program.cs file from the DelegateEx03Console project to the new project.
5. Edit the copied version of Program.cs to have the namespace AnonymousMethodConsole.
6. Change the first call to Console.WriteLine in the Main method to print "Anonymous Methods" to the console.

7. Set AnonymousMethodConsole to be your start-up project and run it to verify that the example runs and prints “Anonymous Methods” as the first line to the console.

Now we can modify the new project’s Main method to use anonymous methods instead of delegates.

Replace the line

```
Transformer<double> doubleTransform = Square;
```

with

```
Transformer<double> doubleTransform =
    delegate(double x)
    {
        return x * x;
    };
```

and delete the version of the Square method that works with doubles.

The blue text shows how to define an ***anonymous method***. It looks pretty similar to a normal method definition, with a parameter list, method body and return statement, though the keyword `delegate` appears where you would normally specify a method name, and there is no need for an access modifier or return type (the return type is implied by the context in which the anonymous method appears). If you run this example, you should see that it still works as before. This achieves our main objective of moving the method implementation to the point in the code where we need it – and notice that we do not need to even give this method a name (hence the moniker *anonymous*).

In fact, instead of explicitly creating the named delegate doubleTransform, you can simply insert the anonymous method declaration in place of the second argument to the Transform method.

Let’s do it this way for the integer array, by replacing

```
Transformers.Transform(myIntegers, Square);
```

with

```
Transformers.Transform(myIntegers,
    delegate(int i)
    {
        return i * i;
    }
    );
```

You can now delete the overload of the Square method for `int`s.

You might take a minute to contemplate all that is happening in this “one line” of code:

- The Transform method’s first argument is an array of type `int`, so the compiler recognizes that the type parameter for the Transform method must be `int`, and the compiler creates a Transform method tailored for working with `int`s.

- The second parameter for the Transform method is `Transformer<T>`, a delegate that is compatible with methods that accept a single parameter of type T and return a value of type T. Since T is `int`, the compiler needs a delegate of type `Transformer<int>`.
- The delegate keyword begins the definition of an anonymous method that is compatible with the `Transformer<int>` delegate, so a method is created at compile-time without giving it a name.
- Since the compiler wants a `Transformer<int>` argument for its second parameter, and we have supplied a method instead, the compiler infers that it needs to instantiate a delegate of type `Transformer<int>` using a constructor and pass in the anonymous method as the parameter to the constructor, so the delegate refers to the anonymous method.

Wow! That's a lot of activity for this one line of code.

To take this one step further, we can make the same modification to the code that uses the Func<T> delegate instead of the Transformer<T> delegate by replacing

```
Transformers.TransformFunc(myDecimals, Square);
```

with

```
Transformers.TransformFunc(myDecimals,
    delegate(decimal d)
    {
        return d * d;
    }
    );
```

Now you can delete the version of the Square method that works with the decimal type.

Run the application and verify that the AnonymousMethodConsole example gives you exactly the same results as the DelegateEx03Console example.

Take a minute to compare the code in two different Main methods. Do you think it is an improvement?

The Anonymous Methods example's primary benefit is that the method body appears in-line at the only place where it is used, so someone reading the code does not have to track down the location of some referenced method to determine that the operation just amounts to taking the square of a number. The downside is that it uses some more advanced C# features that take a while to learn and understand. Anonymous methods are really only suitable if the method you are implementing is only used in this particular method call – we cannot re-use the method effectively, since it doesn't have a name. It also only makes sense when the anonymous method logic is pretty simple – if the method is too long or elaborate, it becomes hard to follow when a large method body is inserted in place of one of the parameters of a method call.

Step 4: Lambda Statements. In version 3.0 of the .Net Framework, Microsoft released an improved version of anonymous methods with a different syntax that is more concise (this seems to be a recurring theme!), called ***lambda statements***.

Add another new Console application to the solution and call it LambdaStatementConsole.

Replace the Program.cs file in the LambdaStatementConsole project with the one from the AnonymousMethodConsole project, change the namespace to "LambdaStatementConsole", change the first Console.WriteLine method to write "Lambda Statements:" to the console, and add a project reference to point to the `DelegateLibrary` project. Set the new console project as your startup project and run it to verify that it works and prints "Lambda Statements" on the first output line to the console. Now we can modify this example to use Lambda Expression syntax.

Replace the line

```
Console.WriteLine("\r\nSquaring doubles with callback:");
Transformer<double> doubleTransform =
        delegate(double x)
        {
            return x * x;
        };

Transformers.Transform(myDoubles, doubleTransform);
```

With the following

```
Console.WriteLine("\r\nSquaring doubles with callback:");
Transformer<double> doubleTransform =
        (double x) =>
        {
            return x * x;
        };

Transformers.Transform(myDoubles, doubleTransform);
```

The code highlighted in blue is a lambda statement, which is equivalent to the corresponding anonymous method in the previous step.

The operator => is the ***lambda operator***, and when reading aloud, you generally use the words "goes to" in place of this operator. On the left side of the operator you will have a parameter list that is pretty much identical to a method's parameter list. On the right side of the operator will be the lambda statement body that is just like a method body and can contain a code block of statements. If the lambda statement is expecting a return value, then the lambda body should include a return statement.

Like parameter lists in a method definition, the parameter list for a lambda statement consists of a comma-separated list of types and local identifiers enclosed in parentheses. Lambda statements, however, do not appear in isolation, but have some context where the compiler can always work out what the expected return type is and usually determine the parameter types.

Consequently, you can (and do) usually omit explicitly specifying the parameter types as in the above example, where I could have used `(x)` in place of `(double x)`. In the special case where there is a single parameter and its type can be inferred, then you can even omit the parentheses. If there are no parameters at all, then you need to use a pair of empty parentheses on the left side of the lambda operator.

To convert the rest of this example to using lambda statements, replace

```
Transformers.Transform(myIntegers,
    delegate(int i)
    {
        return i * i;
    }
    );
```

with

```
Transformers.Transform(myIntegers,
    i =>
    {
        return i * i;
    }
    );
```

And then replace

```
Transformers.TransformFunc(myDecimals,
    delegate(decimal d)
    {
        return d * d;
    }
    );
```

with

```
Transformers.TransformFunc(myDecimals,
    d =>
    {
        return d * d;
    }
    );
```

You should be able to run this example now and get exactly the same results as the prior two examples. This is really just a slight improvement over the anonymous method syntax from .Net 2.0. It's debatable whether this is really much of an improvement over anonymous methods at all for basic usage like this, but it does support a feature called variable lifting that we will explore toward the end of this exercise.

Step 5: Lambda Expressions. Frequently, the in-line method you need to implement is really a single line of code that could be written as a single statement or return expression. In such cases, you can use a shorter form of lambda that replaces the method body of a lambda statement with a single expression or statement.

Create a new ConsoleApplication project called LambdaExpressionConsole, add a reference to the new project that references the DelegateLibrary project, and replace the Program.cs file with a copy of the Program.cs file from Step 4. Fix the namespace and change the first line that writes to the console to write “Lambda Expressions:” instead of “Lambda Statements;”. Then set the new project as your start-up project and run it to make sure it still works.

Now we can replace the Lambda Statements with Lambda expressions, as follows:

Replace

```
Transformer<double> doubleTransform =
        (double x) =>
        {
            return x * x;
        };
```

with

```
Transformer<double> doubleTransform = x => x * x;
```

replace

```
Transformers.Transform(myIntegers,
    i =>
    {
        return i * i;
    }
    );
```

with

```
Transformers.Transform(myIntegers, i => i * i);
```

and replace

```
Transformers.TransformFunc(myDecimals,
    d =>
    {
        return d * d;
    }
    );
```

with

```
Transformers.TransformFunc(myDecimals, d => d * d );
```

You should be able to uncomment the line

```
Transformers.TransformFunc(myDecimals, d => d * d );
```

and comment out the other calls to Run() methods and run the application to verify that this works.

As you can see, when you can use a lambda expression to represent the method body, the code becomes ***extremely concise and very easy to read and understand***. Once you become accustomed to the lambda expression syntax, you tend to forget about all of the work that the compiler is doing to infer the parameter and return types, create the anonymous function, instantiate the delegate, etc., and this is a good thing, since it allows you to focus on your logic, rather than a bunch of complicated C# syntax.

A few things to note:

Lambda expressions are most suitable when the logic of method can be clearly expressed in one simple expression. Lambda statements are called for when the logic of the method is better represented as multiple statements.

In my example the difference between the Transform method and the TransformFunc method is only in the exact type of the second argument to the method. Once we start using lambdas, the distinction between using a Transform<T> delegate and a Func<T, T> delegate doesn't matter, so it isn't really necessary for us to create our own Transform<T> delegate at all.

Step 5: Exercises. To see if you understand the various ways of in-lining methods, I included an ExercisesConsole application project. Set it to be the startup project and look at the Program.cs file source code.

I've create a few static methods and code in a Main() method to create the corresponding delegates and invoke them. See if you can insert the appropriate code to instantiate the same logic using Anonymous Methods, Lambda Statements and Lambda Expression by replacing each occurrence of `/* Insert here */` with the appropriate code.

Step 6: Using a Lambda Expression with the List<T>.Sort() method. Lambda expressions are very convenient when working with FCL methods that take delegate arguments, since you frequently just need to use a line or two of logic to implement the callback. I've also copied DelegateEx4 from the previous lab into this solution.

Create a `LambdaSortConsole` console application and copy the Program.cs file from the `DelegateEx4Console` project to the new project. Fix the namespace, change the first WriteLine method to write "Lambda Expression Sorting:" instead of "Delegate Example 3:", add a reference to the Talent.Domain and Talent.Domain.TestData projects, set the LambdaSortConsole project to be the startup project and run it to make sure that it still works.

See if you can modify the line

```
store.People.Sort(FirstNameComparison);
```

to use a lambda expression instead of the EmployeeNumberComparison method. You should be able to get rid of the FirstNameComparison static method now, since the lambda expression replaces it.

If you get this working, you can look up the documentation for the List<T>.Find method and see if you can write a bit of code that searches emps for an employee with the first name of "Dudley" and then prints the employee name to the console.

Step 7: Variable Capture and the List<T>.Find() method. One feature of Lambdas and Anonymous Methods that is very helpful is called variable capture (or sometimes variable lifting). Inside the body of a method, you normally only have access to variables that are scoped to the method body itself, including local variables and instance and static variables within the method's containing class. However, when using lambdas, the body of the lambda expression or statement can actually reference variables that are scoped to the method body containing the lambda (sometimes called outer variables).

To illustrate, we can add some additional code to the Main method that uses the Find method for the List<T> class:

```
Console.WriteLine("Example of Variable Capture with Lambdas");
while(true)
{
    Console.WriteLine("Enter a partial name and <Enter> to search, or just <Enter> to 
quit:");
    string criterion = Console.ReadLine();
    if (String.IsNullOrWhiteSpace(criterion)) break;

    Person person = store.People
        .Find(p => p.FirstLastName.ToUpper().Contains(criterion.ToUpper()));
    if (person == null)
    {
        Console.WriteLine("(No matches)");
    }
    else
    {
        Console.WriteLine(person.FirstLastName);
    }
}
```

Lookup the Find method in the MSDN documentation and you will see that the syntax for it is:

```
public T Find(
         Predicate<T> match
)
```

Where Predicate<T> is a delegate defined in the FCL.

The key point here is the lambda expression in blue text. To be compliant with the delegate type that the Find method needs, Predicate<EmployeeBase>, the lambda expression must accept an EmployeeBase parameter and return a boolean value (true if the employee matches a condition, false if it doesn't). What the lambda needs to do, though, is compare the FirstLastName property to the variable `criterion` which is scoped to the Main() method, and would not normally be available in the method body of the lambda. However, the compiler employs some trickery to determine if variables that are scoped to the Run method are required by the lambda and then makes them available to the lambda expression. How the CLR implements this is explained in the blog http://blogs.msdn.com/b/matt/archive/2008/03/01/understanding-variable-capturing-in-c.aspx if you are interested in what the compiler does to support this. This is pretty cumbersome to try to duplicate with methods alone.

Lab Summary

This lab explored various ways of creating small methods in an "in-line" fashion for one-time use within a particular context. Each of the techniques demonstrated falls under the heading of "anonymous functions":

- Anonymous Methods
- Lambda Statements, and
- Lambda Expressions

Of these, Lambda Statements are preferred over the older Anonymous methods, unless you are maintaining code written in C# version 2.0. Lambda expressions are normally more compact and easier to read than Lambda Statements for very simple logic that can be clearly expressed in a single expression. Lambdas tend to make your code easier to read and clutter in your code, since you do not need to have simple functions scattered around your code base.

In a later lesson, we look at LINQ, which generally involves using many delegates, and lambda expressions are used extensively in LINQ to make LINQ queries quite readable.

You may have noticed the recurring theme in the last few labs of trying to make code more concise. When you first look at delegates, they appear to be introducing a lot of complicated syntax which makes code harder to read. However, once you get accustomed to reading code involving lambdas, you should find that it becomes relatively easy to read and allows you to think about your code at a somewhat more abstract level.

Finished Solution

You can download my finished solution to this exercise from https://github.com/entrotech/Lab06-Lambdas.