Lambda Expressions

One way where lambda expressions are used is to assign a lambda expression to a delegate type: implement code inline. Lambda expressions can be used whenever you have a delegate parameter type. The previous example using anonymous methods is modified here to use a lambda expression.

```
using System;
using static System.Console;
namespace Wrox.ProCSharp.Delegates
{
  class Program
  {
    static void Main()
    {
      string mid =", middle part,";

      Func<string, string> lambda = param =>
         {
            param += mid;
            param +=" and this was added to the string.";
            return param;
         };

      WriteLine(lambda("Start of string"));
    }
  }
}
```

The left side of the lambda operator, =>, lists the parameters needed. The right side following the lambda operator defines the implementation of the method assigned to the variable `lambda`.

Parameters

With lambda expressions there are several ways to define parameters. If there's only one parameter, just the name of the parameter is enough. The following lambda expression uses the parameter named `s`. Because the delegate type defines a `string` parameter, `s` is of type `string`. The implementation invokes the `String.Format` method to return a string that is finally written to the console when the delegate is invoked: `change uppercase TEST` (code file `LambdaExpressions/Program.cs`):

```
Func<string, string> oneParam = s =>

          $"change uppercase {s.ToUpper()}";

WriteLine(oneParam("test"));
```

If a delegate uses more than one parameter, you can combine the parameter names inside brackets. Here, the parameters `x` and `y` are of type `double` as defined by the `Func<double, double, double>` delegate:

```
Func<double, double, double> twoParams = (x, y) => x * y;

WriteLine(twoParams(3, 2));
```

For convenience, you can add the parameter types to the variable names inside the brackets. If the compiler can't match an overloaded version, using parameter types can help resolve the matching delegate:

```
Func<double, double, double> twoParamsWithTypes = (double x, double y) => x *
y;

WriteLine(twoParamsWithTypes(4, 2));
```

Multiple Code Lines

If the lambda expression consists of a single statement, a method block with curly brackets and a `return` statement are not needed. There's an implicit `return` added by the compiler:

```
Func<double, double> square = x => x * x;
```

It's completely legal to add curly brackets, a `return` statement, and semicolons. Usually it's just easier to read without them:

```
Func<double, double> square = x =>
  {
    return x * x;
  }
```

However, if you need multiple statements in the implementation of the lambda expression, curly brackets and the `return`statement are required:

```
Func<string, string> lambda = param =>
  {
    param += mid;
    param +=" and this was added to the string.";
    return param;
  };
```

Closures

With lambda expressions you can access variables outside the block of the lambda expression. This is known by the term *closure*. Closures are a great feature, but they can also be very dangerous if not used correctly.

In the following example, a lambda expression of type `Func<int, int>` requires one `int` parameter and returns an `int`. The parameter for the lambda expression is defined with the variable `x`. The implementation also accesses the variable `someVal`, which is outside the lambda expression. As long as you do not assume that the lambda expression creates a new method that is used later when `f` is invoked, this might not look confusing at all. Looking at this code block, the returned value calling `f` should be the value from `x` plus 5, but this might not be the case:

```
int someVal = 5;

Func<int, int> f = x => x + someVal;
```

Assuming the variable `someVal` is later changed, and then the lambda expression is invoked, the new value of `someVal` is used. The result here of invoking `f(3)` is `10`:

```
someVal = 7;

WriteLine(f(3));
```

Similarly, when you're changing the value of a closure variable within the lambda expression, you can access the changed value outside of the lambda expression.

Now, you might wonder how it is possible at all to access variables outside of the lambda expression from within the lambda expression. To understand this, consider what the compiler does when you define a lambda expression. With the lambda expression `x => x + someVal`, the compiler creates an anonymous class that has a constructor to pass the outer variable. The constructor depends on how many variables you access from the outside. With this simple example, the constructor accepts an `int`. The anonymous class contains an anonymous method that has the implementation as defined by the lambda expression, with the parameters and return type:

```
public class AnonymousClass
{
  private int someVal;
  public AnonymousClass(int someVal)
  {
    this.someVal = someVal;
  }
  public int AnonymousMethod(int x) => x + someVal;
}
```

Using the lambda expression and invoking the method creates an instance of the anonymous class and passes the value of the variable from the time when the call is made.

NOTE *In case you are using closures with multiple threads, you can get into concurrency conflicts. It's best to only use immutable types for closures. This way it's guaranteed the value can't change, and synchronization is not needed.*

NOTE *You can use lambda expressions anywhere the type is a delegate. Another use of lambda expressions is when the type is* `Expression` *or* `Expression<T>`*., in which case the compiler creates an expression tree. This feature is discussed in Chapter 13, "Language Integrated Query."*