Understanding delegates

A delegate is a reference to a method.

Declaring Delegates

The syntax for declaring delegates looks like this:

```
delegate void IntMethodInvoker(int x);
delegate double TwoLongsOp(long first, long second);
delegate string GetAString();
```

Delegate assignment

```
Processor p = new Processor();
delegate ... performCalculationDelegate ...;
performCalculationDelegate = p.performCalculation;
```

The statement that assigns the method reference to the delegate does not run the method at that point; there are no parentheses after the method name, and you do not specify any parameters (if the method takes them).

Invoke the delegate

The application can subsequently invoke the method through the delegate.

```
performCalculationDelegate();
```

Sample delegate example

```
class MathOperations
{
  public static double MultiplyByTwo(double value) => value * 2;

  public static double Square(double value) => value * value;
}
```

You invoke these methods as follows

```
using static System.Console;

namespace example.Delegates
{
  delegate double DoubleOp(double x);

  class Program
  {
    static void Main()
    {
      DoubleOp[] operations =
      {
        MathOperations.MultiplyByTwo,
        MathOperations.Square
      };

      for (int i=0; i < operations.Length; i++)
      {
        WriteLine($"Using operations[{i}]:);
        ProcessAndDisplayNumber(operations[i], 2.0);
        ProcessAndDisplayNumber(operations[i], 7.94);
        ProcessAndDisplayNumber(operations[i], 1.414);
        WriteLine();
      }
    }

    static void ProcessAndDisplayNumber(DoubleOp action, double value)
    {
      double result = action(value);
      WriteLine($"Value is {value}, result of operation is {result}");
    }
  }
}
```

Action<T> and Func<T> Delegates

- Instead of defining a new delegate type with every parameter and return type, you can use the `Action<T>` and `Func<T>` delegates.
- The generic `Action<T>`delegate is meant to reference a method with `void` return.
- This delegate class exists in different variants so that you can pass up to 16 different parameter types. `Action<in T1, in T2>` for a method with two parameters
- `Func<T>` allows you to invoke methods with a return type.
- Similar to `Action<T>`, `Func<T>` is defined in different variants to pass up to 16 parameter types and a return type.
- `Func<out TResult>` is the delegate type to invoke a method with a return type and without parameters.
- `Func<in T1, in T2, in T3, in T4, out TResult>`is for a method with four parameters.

The example below declared a delegate with a `double` parameter and a `double` return type:

```
delegate double DoubleOp(double x);
```

Instead of declaring the custom delegate `DoubleOp` you can use the `Func<in T, out TResult>` delegate.

You can declare a variable of the delegate type or, as shown here, an array of the delegate type:

```
<double, double>[] operations =
{
  MathOperations.MultiplyByTwo,
  MathOperations.Square
};
```

and use it with the `ProcessAndDisplayNumber` method as a parameter:

```
static void ProcessAndDisplayNumber(Func<double,
double> action,double value)
{
  double result = action(value);
  WriteLine($"Value is {value}, result of
operation is {result}");
}
```

Multicast Delegates

It is possible for a delegate to wrap more than one method. Such a delegate is known as a multicast delegate.

```
class Program
{
    static void Main()
    {
      Action<double> operations =
MathOperations.MultiplyByTwo;
      operations += MathOperations.Square;
```

In the earlier example, you wanted to store references to two methods, so you instantiated an array of delegates. Here, you simply add both operations into the same multicast delegate.

```
Action<double> operation1 =
MathOperations.MultiplyByTwo;

Action<double> operation2 =
MathOperations.Square;

Action<double> operations = operation1 +
operation2;
```

Complete example

```
class MathOperations
{
  public static void MultiplyByTwo(double value)
  {
    double result = value * 2;
    WriteLine($"Multiplying by 2: {value} gives
{result}");
  }

  public static void Square(double value)
  {
    double result = value * value;
    WriteLine($"Squaring: {value} gives
{result}");
  }
}
```

To accommodate this change, you also have to rewrite `ProcessAndDisplayNumber`.

```
static void
ProcessAndDisplayNumber(Action<double> action,
double value)
{
  WriteLine();
  WriteLine($"ProcessAndDisplayNumber called
with value = {value}");
  action(value);
}
```

Now you can try out your multicast delegate:

```
static void Main()
{
  Action<double> operations =
MathOperations.MultiplyByTwo;
  operations += MathOperations.Square;
```

```
  ProcessAndDisplayNumber(operations, 2.0);
  ProcessAndDisplayNumber(operations, 7.94);
  ProcessAndDisplayNumber(operations, 1.414);
  WriteLine();
}
```

Invoking multiple methods by one delegate might cause an even bigger problem. The multicast delegate contains a collection of delegates to invoke one after the other. If one of the methods invoked by a delegate throws an exception, the complete iteration stops.

```
using System;
using static System.Console;

namespace ProCSharp.Delegates
{
  class Program
  {
    static void One()
    {
      WriteLine("One");
      throw new Exception("Error in one");
    }

    static void Two()
    {
      WriteLine("Two");
    }
```

In the `Main` method, delegate `d1` is created to reference method `One`; next, the address of method `Two` is added to the same delegate. `d1` is invoked to call both methods. The exception is caught in a `try/catch` block:

```
    static void Main()
    {
      Action d1 = One;
      d1 += Two;

      try
      {
        d1();
      }
      catch (Exception)
      {
        WriteLine("Exception caught");
      }
    }
  }
}
```

Only the first method is invoked by the delegate. Because the first method throws an exception, iterating the delegates stops here and method `Two` is never invoked. The result might differ because the order of calling the methods is not defined:

```
One
Exception Caught
```

In such a scenario, you can avoid the problem by iterating the list on your own. The `Delegate` class defines the method `GetInvocationList` that returns an array of `Delegate` objects. You can now use this delegate to invoke the methods associated with them directly, catch exceptions, and continue with the next iteration:

```
static void Main()
{
  Action d1 = One;
  d1 += Two;
```

```
  Delegate[] delegates = d1.GetInvocationList();
  foreach (Action d in delegates)
  {
    try
    {
      d();
    }
    catch (Exception)
    {
      WriteLine("Exception caught");
    }
  }
}
```

When you run the application with the code changes, you can see that the iteration continues with the next method after the exception is caught:

```
One
Exception caught
Two
```

Anonymous Methods

There is another way to use delegates—with *anonymous methods*. An anonymous method is a block of code that is used as the parameter for the delegate.

- The syntax for defining a delegate with an anonymous method doesn't change. It's when the delegate is instantiated that things change.

```
using static System.Console;
using System;
```

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

      Func<string, string> anonDel =
delegate(string param)
      {
        param += mid;
        param +=" and this was added to the
string.";
        return param;
      };
      WriteLine(anonDel("Start of string"));

    }
  }
}
```

The benefit of using anonymous methods is that it reduces the amount of code you have to write. You don't need to define a method just to use it with a delegate. This becomes evident when you define the delegate for an event. , and it helps reduce the complexity of code, especially where several events are defined.

You must follow a couple of rules when using anonymous methods. You can't have a jump statement (`break`, `goto`, or `continue`) in an anonymous method that has a target outside of the anonymous method. The reverse is also true: A jump statement outside the anonymous method cannot have a target inside the anonymous method.

Unsafe code cannot be accessed inside an anonymous method, and the `ref` and `out` parameters that are used outside of the anonymous method cannot be accessed. Other variables defined outside of the anonymous method can be used.

If you have to write the same functionality more than once, don't use anonymous methods. In this case, instead of duplicating the code, write a named method. You have to write it only once and reference it by its name.

Lambda Expressions

One way where lambda expressions are used is to assign a lambda expression to a delegate type: implement code inline.

The previous example using anonymous methods is modified here to use a lambda expression.

```
using System;
using static System.Console;
namespace 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.

```
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.

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.