Anonymous Methods

Up to this point, a method must already exist for the delegate to work (that is, the delegate is defined with the same signature as the method(s) it will be used with). However, 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. The following simple console application shows how using an anonymous method can work (code file `AnonymousMethods/Program.cs`):

```
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 delegate `Func<string, string>` takes a single string parameter and returns a string. `anonDel` is a variable of this delegate type. Instead of assigning the name of a method to this variable, a simple block of code is used, prefixed by the delegate keyword, followed by a string parameter.
As you can see, the block of code uses a method-level string variable, `mid`, which is defined outside of the anonymous method and adds it to the parameter that was passed in. The code then returns the string value. When the delegate is called, a string is passed in as the parameter and the returned string is output to the console.
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 (events are discussed later in this chapter), and it helps reduce the complexity of code, especially where several events are defined. With anonymous methods, the code does not perform faster. The compiler still defines a method; the method just has an automatically assigned name that you don’t need to know.

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.
NOTE *The syntax for anonymous methods was introduced with* C# 2. *With new programs you really don't need this syntax anymore because lambda expressions (explained in the next section) offer the same—and more—functionality. However, you'll find the syntax for anonymous methods in many places in existing source code, which is why it's good to know it.*
Lambda expressions have been available since C# *3.*