Delegates

Executable code typically exists as a method belonging to a static, or instance of a struct or class. A delegate is another reference type that represents a method, similar to a function pointer in other languages. However, delegates are type-safe and secure, providing object-oriented functions.

Delegates in C# and VB.NET inherit from `System.MulticastDelegate` (which inherits from `System.Delegate` and, in turn, from `System.Object`). This class provides a linked list of delegates which are invoked synchronously in order (Listing 4.40).

```
public delegate void Function();

public void WriteInformation()
{
 Debug.WriteLine("Got Here");
}

public void CallFunction()
{
 Function function = new Function(WriteInformation);
 function();
}
```

Listing 4.40: Declare a delegate and associate a method with it.

Parameters can be added to a delegate, which are then necessary for the invocation of the method the delegate is instantiated with, as well as the delegate itself (Listing 4.41).

```
public delegate void Function(string info);

public void WriteInformation(string info)
{
 Debug.WriteLine(info);
}

public void CallFunction()
{
 Function function = new Function(WriteInformation);
 function("function called");
}
```

Listing 4.41: Declare a delegate with parameters and associate a method with it.

Delegates can be combined by calling the `Delegate.Combine` method, although the semantics of the delegate are then lost and it must be invoked through the `Dynamic-Invoke()` method, as shown in Listing 4.42.

```
public delegate void Function(string info);

public void WriteInformation(string info)
{
 Debug.WriteLine(info);
}

public void WriteToConsole(string info)
{
 Console.WriteLine(info);
}

public void CallFunction()
{
 Function function = new Function(WriteInformation);
 Function another = new Function(WriteToConsole);
 var del = Delegate.Combine(function, another);
 del.DynamicInvoke("multicast");
}
```

Listing 4.42: Associating multiple methods with a delegate.

Be aware that when a multicast delegate is invoked, the delegates in the invocation list are called synchronously in the order in which they appear. If an error occurs during execution of the list, an exception is thrown.

This ability is useful for writing reusable delegates, but oftentimes delegates are scoped within a method. The concept of anonymous methods shown in Listing 4.43 was introduced as an alternative to named methods.

```
public delegate void Function(string info);

public void CallFunction()
{
 Function function = delegate(string info) { Debug.WriteLine(info); };
 function("This method is anonymous");
}
```

Listing 4.43: Associating an anonymous method with a delegate.

More innovations in the .NET languages took place, and anonymous functions were introduced. Lambda expressions further reduced the amount of typing necessary to create a delegate. In addition, standardized, generic delegate types were introduced to the .NET framework to easily create delegates when necessary. Delegates without return types are known as `Actions`, and those with return types are known as `Funcs`. The last generic parameter of a `Func` type is the `return` type.

```
Action<string> function = info => Debug.WriteLine(info);
function("Created from a lambda");
```

Listing 4.44: A simple lambda action.

Despite the progression of delegate usage in the .NET languages, the story behind the scenes remains largely the same. The compiler creates a static delegate definition and a named method, and places the code contained within the lambda expression inside of that named method. It then creates the delegate, if it does not already exist, and calls it. Anonymous methods and functions are really just syntactic sugar.

A common use of delegates is for event handlers, as they allow you to loosely couple functionality through an event model. The classic example of this is performing an action when a button is clicked, shown in Listing 4.45 and Figure 4.7.

```
public MainWindow()
{
 InitializeComponent();
 button.Click += (sender, e) => MessageBox.Show("Under the Hood");
}
```

Listing 4.45: Wiring up an event handler to an anonymous method.

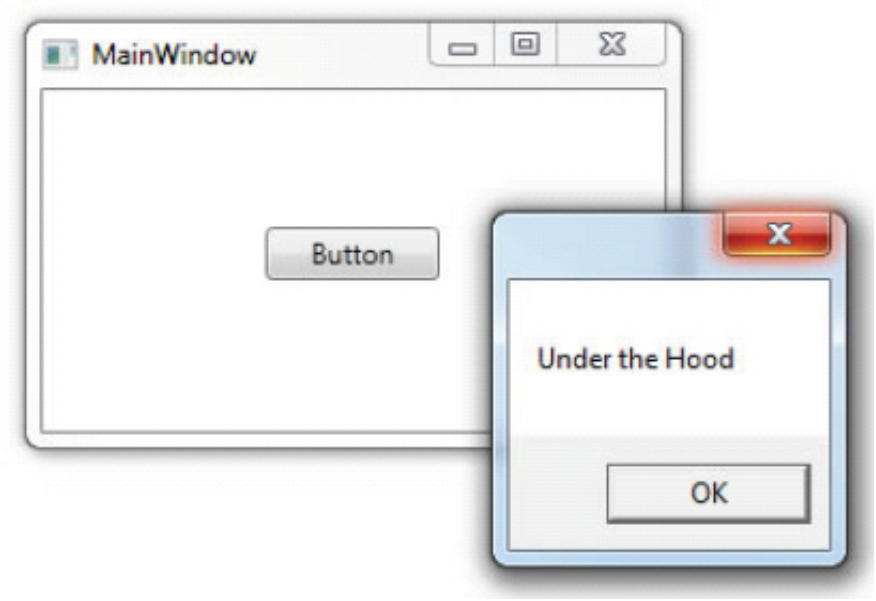

Figure 4.7: The outcome from Listing 4.45.

Notice the operator that adds the lambda expression to the `Click` event. Event handlers are added to an event list, ready to be invoked, and it is important to release the delegates in the event list, or set the event to null when disposing of the object (and classes that implement events should implement `IDisposable`). This might seem like stating the obvious, and the situation isn't so bad when only one object has a reference to the delegate. However, if more than one object has a reference to the same delegate, they will all be kept alive until all the objects with a reference to the same delegate are released. We will explore this more fully in Chapter 5.

Closures

Variables that are created outside of an anonymous method but used within it are captured in a closure. This allows the delegate to have access to the value even when the variable goes out of scope.

```
string str = "Original";
Action writeLine = () => Console.WriteLine(str);
writeLine();

str ="Changed";
writeLine();
```

Listing 4.46: A very simple closure.

From reading the code in Listing 4.46 it is expected that it will first write `"Original"` to the screen, followed by `"Changed"`. That is exactly how it works, which perhaps seems odd in light of the fact that `string` is immutable and it has already been passed into the delegate.

What happens is that the compiler does the heavy lifting for you, creating a class and assigning the values to it. This will actually affect you any time you use an outside variable inside of an anonymous method or function. Consider the slightly more complex closure in Listing 4.47.

```
public Person WhoReportsTo (List<Employee > data, Person targetEmployee)
{
 return data.Find(d => d.ReportsTo == targetEmployee);
}
```

Listing 4.47: A more complex example with closures.

Once the compiler works its magic, we get the code in Listing 4.48.

```
[CompilerGenerated]
private sealed class <>c__DisplayClass4
{
 // Fields
 public Person targetEmployee;

 // Methods
 public bool <WhoReportsTo>b__3(Employee d)
 {
  return (d.ReportsTo == this.targetEmployee);
 }
}
```

Listing 4.48: The dynamic class generated enclosing the captured variables.

The anonymous method is actually "enclosed" in a dynamic compiler-generated class, which includes a member variable for every external variable that the anonymous method references. This generated class and all enclosed variables will stay alive as long as the delegate is accessible. This extended lifetime can dramatically delay when variables are eligible for garbage collection, and thus create situations that look like a memory leak.

Avoid closures with memory intensive variables.