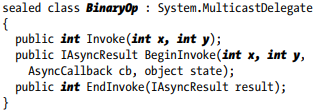
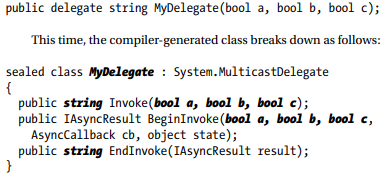
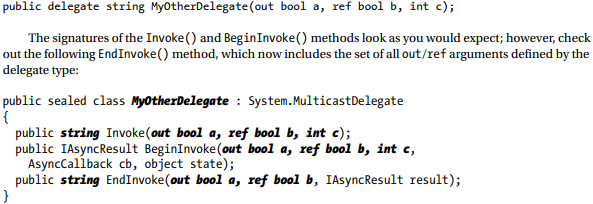
**Delegates, Events, and Lambda Expressions** Up to this point in the text, most of the applications you developed added various bits of code to Main(), which, in some way or another, sent requests to a given object. However, many applications require that an object be able to communicate back to the entity that created it using a callback mechanism. While callback mechanisms can be used in any application, they are especially critical for graphical user interfaces in that controls (such as a button) need to invoke external methods under the correct circumstances (when the button is clicked, when the mouse enters the button surface, and so forth). Under the .NET platform, the delegate type is the preferred means of defining and responding to callbacks within applications. Essentially, the .NET delegate type is a type-safe object that “points to” a method or a list of methods that can be invoked at a later time. Unlike a traditional C++ function pointer, however, .NET delegates are classes that have built-in support for multicasting and asynchronous method invocation. In this chapter, you will learn how to create and manipulate delegate types, and then you’ll investigate the C# event keyword, which streamlines the process of working with delegate types. Along the way, you will also examine several delegate- and event-centric language features of C#, including anonymous methods and method group conversions. I wrap up this chapter by examining lambda expressions. Using the C# lambda operator (=>), you can specify a block of code statements (and the parameters to pass to those code statements) wherever a strongly typed delegate is required. As you will see, a lambda expression is little more than an anonymous method in disguise and provides a simplified approach to working with delegates. In addition, this same operation (as of .NET 4.6) can be used to implement a single-statement method or property using a concise syntax.

**Defining a Delegate Type in C#** When you want to create a delegate type in C#, you use the delegate keyword. The name of your delegate type can be whatever you desire.

As you can see, the compiler-generated BinaryOp class defines three public methods. Invoke() is perhaps the key method, as it is used to invoke each method maintained by the delegate object in a synchronous manner, meaning the caller must wait for the call to complete before continuing on its way. Strangely enough, the synchronous Invoke() method may not need to be called explicitly from your C# code. As you will see in just a bit, Invoke() is called behind the scenes when you use the appropriate C# syntax. BeginInvoke() and EndInvoke() provide the ability to call the current method asynchronously on a separate thread of execution. If you have a background in multithreading, you know that one of the most common reasons developers create secondary threads of execution is to invoke methods that require time to complete. Although the .NET base class libraries supply several namespaces devoted to multithreaded and parallel programming, delegates provide this functionality out of the box.

