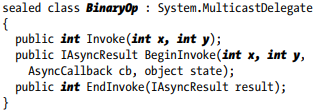
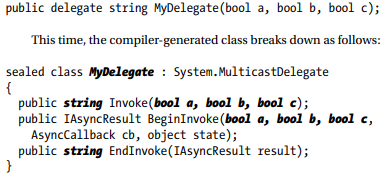
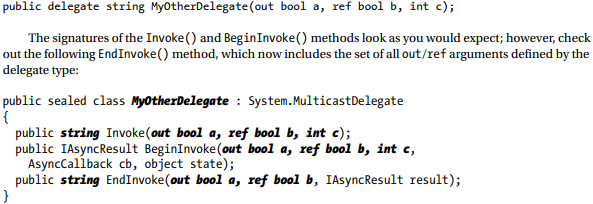
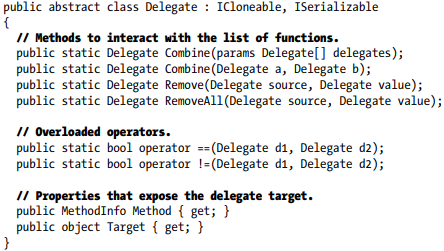
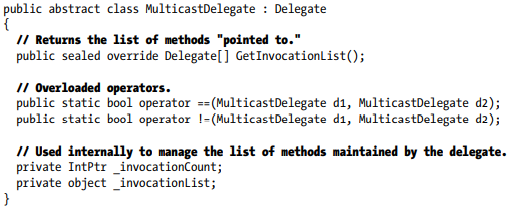
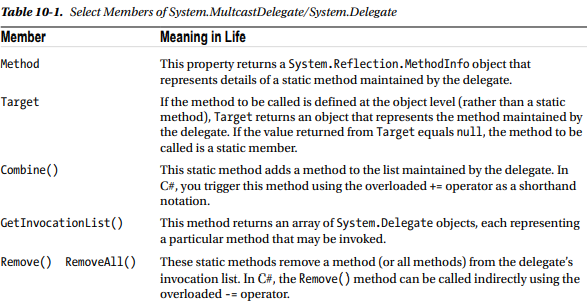
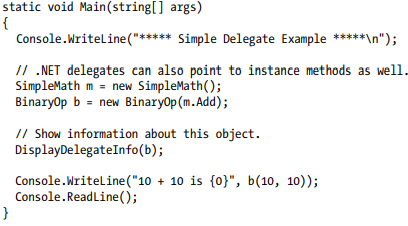
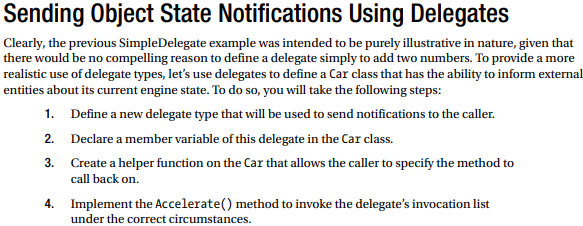
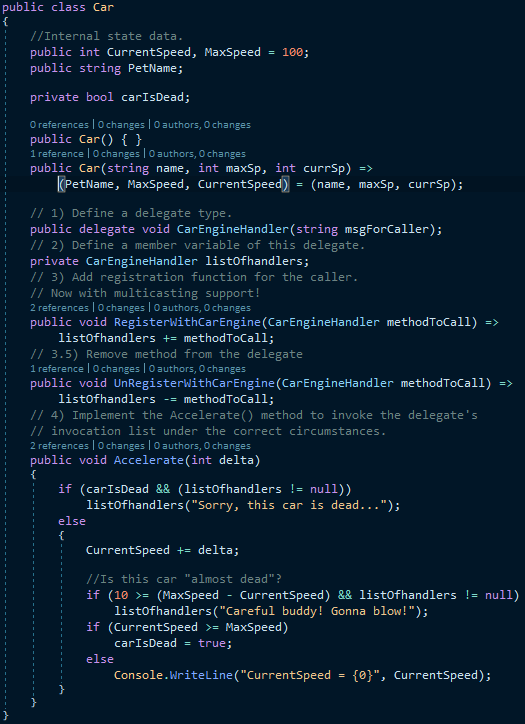
**Delegates, Events, and Lambda Expressions** Up to this point in the text, most of the applications you developed added various bits of code to Main(), which, in some way or another, sent requests to a given object. However, many applications require that an object be able to communicate back to the 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.

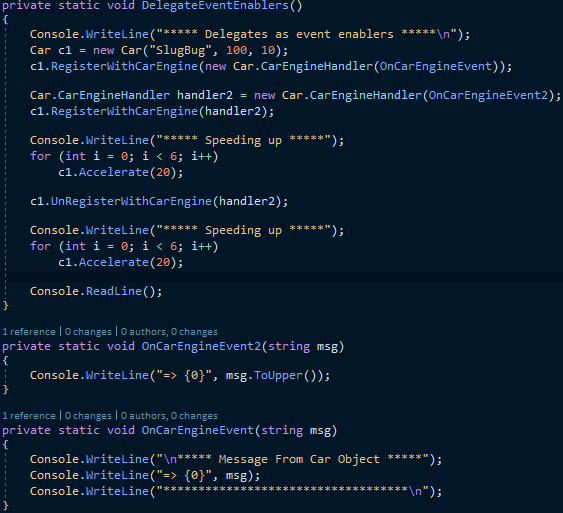
**Defining a Delegate Type in C#** When you want to create a delegate type in C#, you use the delegate keyword. The name of your delegate type can be whatever you desire.

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.

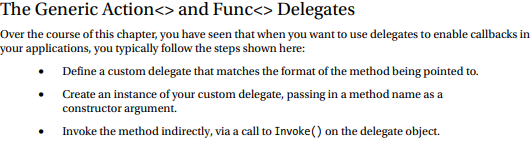
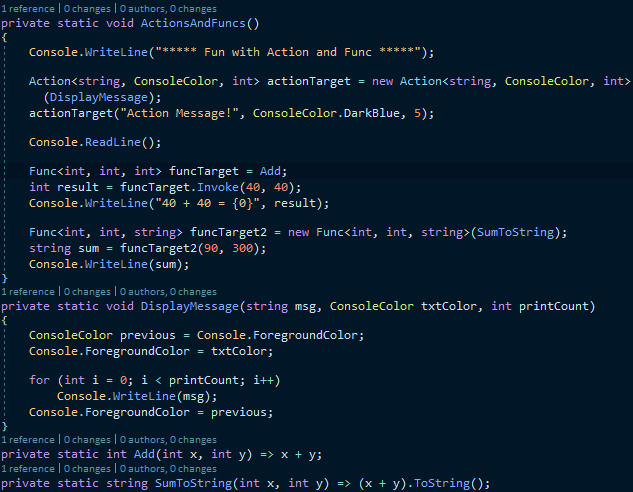




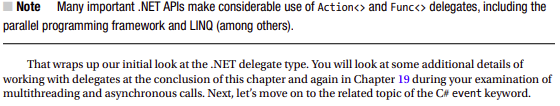
 **Enabling Multicasting** Recall that .NET delegates have the built-in ability to multicast. In other words, a delegate object can maintain a list of methods to call, rather than just a single method. When you want to add multiple methods to a delegate object, you simply use the overloaded += operator, rather than a direct assignment. T



1. Declare delegate type
2. Declare delegate variable/instance
3. Instantiate it with a new type, parameter method token
4. Invoke it by providing it arguments

**Understanding Generic Delegates** In the previous chapter, I mentioned that C# allows you to define generic delegate types. For example, assume you want to define a delegate type that can call any method returning void and receiving a single parameter. If the argument in question may differ, you could model this using a type parameter.

In any case, given that Action<> and Func<> can save you the step of manually defining a custom delegate, you might be wondering if you should use them all the time. The answer, like so many aspects of programming, is “it depends.” In many cases, Action<> and Func<> will be the preferred course of action (no pun intended). However, if you need a delegate that has a custom name that you feel helps better capture your problem domain, building a custom delegate is as simple as a single code statement. You’ll see both approaches as you work over the remainder of this text.



**Understanding C# Events** Delegates are fairly interesting constructs in that they enable objects in memory to engage in a two-way conversation. However, working with delegates in the raw can entail the creation of some boilerplate code (defining the delegate, declaring necessary member variables, and creating custom registration and unregistration methods to preserve encapsulation, etc.). Moreover, when you use delegates in the raw as your application’s callback mechanism, if you do not define a class’s delegate member variables as private, the caller will have direct access to the delegate objects. In this case, the caller could reassign the variable to a new delegate object (effectively deleting the current list of functions to call), and, worse yet, the caller would be able to directly invoke the delegate’s invocation list.

**The C# event Keyword** As a shortcut, so you don’t have to build custom methods to add or remove methods to a delegate’s invocation list, C# provides the event keyword. When the compiler processes the event keyword, you are automatically provided with registration and unregistration methods, as well as any necessary member variables for your delegate types. These delegate member variables are always declared private, and, therefore, they are not directly exposed from the object firing the event. To be sure, the event keyword can be used to simplify how a custom class sends out notifications to external objects. Defining an event is a two-step process. First, you need to define a delegate type (or reuse an existing one) that will hold the list of methods to be called when the even

**Creating Custom Event Arguments** Truth be told, there is one final enhancement you could make to the current iteration of the Car class that mirrors Microsoft’s recommended event pattern. As you begin to explore the events sent by a given type in the base class libraries, you will find that the first parameter of the underlying delegate is a System.Object, while the second parameter is a descendant of System.EventArgs. The System.Object argument represents a reference to the object that sent the event (such as the Car), while the second parameter represents information regarding the event at hand. The System.EventArgs base class represents an event that is not sending any custom information