# Understanding C# Delegates

**The C# event Keyword**

**What are Delegates?**

Most of the applications, 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.

* Under the .NET Core platform, the delegate type is the preferred means of defining and responding to callbacks within applications.
* Essentially, the .NET Core delegate type is a type-safe object (means if you attempt to create a delegate object pointing to a method that does not match the pattern, you receive a compile-time error) that “points to” a method or a list of methods that can be invoked later.
* Delegates are classes that have built-in support for multicasting. In other words, a delegate object can maintain a list of methods to call, rather than just a single method.
* Delegates are interesting constructs, in that they enable objects in memory to engage in a two-way conversation.

In prior versions of .NET, delegates exposed asynchronous method invocation with **BeginInvoke()**/**EndInvoke()**. While these are still generated by the compiler, they are not supported under.NET Core. This is because the IAsyncResult()/BeginInvoke() pattern used by delegates has been replaced by the task-based **async** pattern.

Specifically, a delegate maintains three important pieces of information.

* The address of the method on which it makes calls
* The parameters (if any) of this method
* The return type (if any) of this method

**Note** .NET Core delegates can point to either static or instance methods.

After a delegate object has been created and given the necessary information, it may dynamically invoke the method(s) it points to at runtime.

**Defining a Delegate Type in C#?**

// This delegate can point to any method, taking two integers and returning an integer.

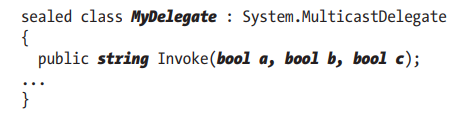
public **delegate** int **BinaryOp**(int x, int y);

When the C# compiler processes delegate types, it automatically generates a **sealed class** deriving from System.MulticastDelegate. This class (in conjunction with its base class, System.Delegate) provides the necessary infrastructure for the delegate to hold onto a list of methods to be invoked later.

As you will see, the compiler-generated BinaryOp class defines three public methods.

* **Invoke()** => 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 its way.
* The synchronous **Invoke()** method may not need to be called explicitly from your C# code. The Invoke() is called behind the scenes when you use the appropriate C# syntax.
* While **BeginInvoke()** and **EndInvoke()** are generated, they are not supported when running your code under .NET Core. You will not receive a compiler error but a runtime error if you use them.

Here is the crux of the compiler-generated BinaryOp class type (bold italic marks the items specified by the defined delegate type):



Delegates can also “point to” methods that contain any number of out or ref parameters (as well as array parameters marked with the params keyword). For example, assume the following delegate type:



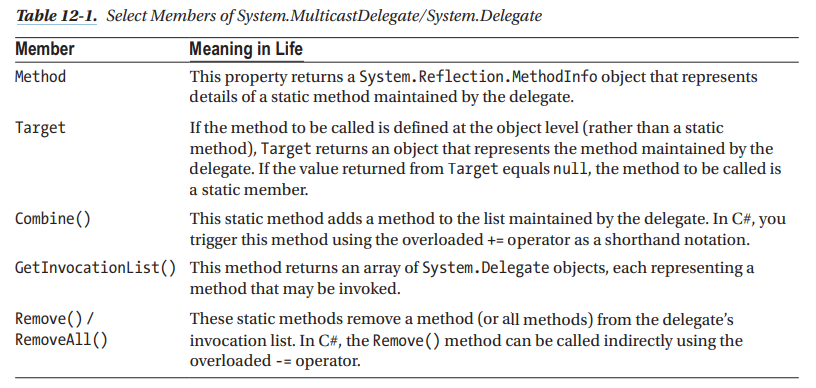
The signature of the Invoke() method looks as you would expect

**The System.MulticastDelegate and System.Delegate Base Classes?**

So, when you build a type using the C# delegate keyword, you are indirectly declaring a class type that derives from System.MulticastDelegate.

This class provides descendants with access to a list that contains the addresses of the methods maintained by the delegate object, as well as several additional methods (and a few overloaded operators) to interact with the invocation list.

NOTE: You can never directly derive from these base classes in your code (it is a compiler error to do so). Nevertheless, when you use the delegate keyword, you have indirectly created a class that “is-a” MulticastDelegate.

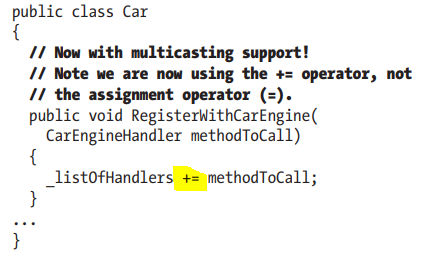


**Sending Object State Notifications Using Delegates**

To provide a more realistic use of delegate types, let’s use delegates to define a Car class that can inform external entities about its current engine state.

**Enabling Multicasting**

* Recall that .NET Core 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.



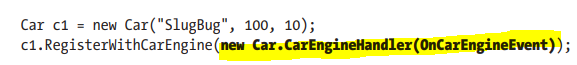
* When you use the **+=** operator on a delegate object, the compiler resolves this to a call on the static **Delegate.Combine()** method. In fact, you could call Delegate.Combine() directly; however, the += operator offers a simpler alternative.

**Removing Targets from a Delegate’s Invocation List**

* The Delegate class also defines a static **Remove()** method that allows a caller to dynamically remove a method from a delegate object’s invocation list.
* This makes it simple to allow the caller to “unsubscribe” from a given notification at runtime.
* C# developers can use the -= operator as a convenient shorthand notation.

**Method Group Conversion Syntax**

As a simplification, C# provides a shortcut termed method group conversion. This feature allows you to supply a direct method name, rather than a delegate object, when calling methods that take delegates as arguments.



Instead



**The Generic Action<> and Func<> Delegates**

Over the course of this chapter, you have seen that when you want to use delegates to enable callbacks in your applications, you typically follow the steps shown here:

1. Define a custom delegate that matches the format of the method being pointed to.
2. Create an instance of your custom delegate, passing in a method name as a constructor argument.
3. Invoke the method indirectly, via a call to Invoke() on the delegate object.

In many cases, you simply want “some delegate” that takes a set of arguments and possibly has a return value other than void. In these cases, you can use the framework’s built-in **Action<>** and **Func<>** delegate types.

**NOTE** Many important .NET Core APIs make considerable use of Action<> and Func<> delegates, including the parallel programming framework and LINQ (among others).

Exposing public delegate members breaks encapsulation, which not only can lead to code that is hard to maintain (and debug) but could also open your application to possible security risks!

Obviously, you would not want to give other applications the power to change what a delegate is pointing to or to invoke the members without your permission. Given this, it is common practice to declare private delegate member variables.

# Understanding C# Events

**Events Under the Hood**

* When the compiler processes the C# event keyword, it generates two hidden methods, one having an add\_ prefix and the other having a remove\_ prefix.
* Each prefix is followed by the name of the C# event. For example, the Exploded event results in two hidden methods named add\_Exploded() and remove\_ Exploded().

**Summary**

* You looked at the C# delegate keyword, which is used to indirectly construct a class derived from System.MulticastDelegate. As you saw, a delegate object maintains the method to call when told to do so.
* You then examined the C# event keyword, which, when used in conjunction with a delegate type, can
* simplify the process of sending your event notifications to waiting callers. As shown via the resulting CIL, the
* The C# event keyword is purely optional, in that it simply saves you some typing time. As well, you have
* seen that the C# 6.0 null conditional operator simplifies how you safely fire events to any interested party.
* Anonymous methods. Using this syntactic construct, you can directly associate a block of code statements to a given event.
* As you have seen, anonymous methods are free to ignore the parameters sent by the event and have access to the “outer variables” of the defining method.
* You also examined a simplified way to register events using method group conversion.
* Finally, you wrapped things up by looking at the C# lambda operator, =>. As shown, this syntax is a great
* shorthand notation for authoring anonymous methods, where a stack of arguments can be passed into a
* group of statements for processing.
* Any method in the .NET Core platform that takes a delegate object as an argument can be substituted with a related lambda expression, which will typically simplify your code base quite a bit.