

Using Delegates (C# Programming Guide)

07/20/2015 • 5 minutes to read •  +6

In this article

[See also](#)

A [delegate](#) is a type that safely encapsulates a method, similar to a function pointer in C and C++. Unlike C function pointers, delegates are object-oriented, type safe, and secure. The type of a delegate is defined by the name of the delegate. The following example declares a delegate named `Del` that can encapsulate a method that takes a [string](#) as an argument and returns [void](#):

C#

 Copy

```
public delegate void Del(string message);
```

A delegate object is normally constructed by providing the name of the method the delegate will wrap, or with an [anonymous function](#). Once a delegate is instantiated, a method call made to the delegate will be passed by the delegate to that method. The parameters passed to the delegate by the caller are passed to the method, and the return value, if any, from the method is returned to the caller by the delegate. This is known as invoking the delegate. An instantiated delegate can be invoked as if it were the wrapped method itself. For example:

C#

 Copy

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}
```

C#


 Copy

```
// Instantiate the delegate.
Del handler = DelegateMethod;


// Call the delegate.
handler("Hello World");
```

Delegate types are derived from the [Delegate](#) class in the .NET Framework. Delegate types are [sealed](#)—they cannot be derived from—and it is not possible to derive custom classes from [Delegate](#). Because the instantiated delegate is an object, it can be passed as a parameter, or assigned to a property. This allows a method to accept a delegate as a parameter, and call the delegate at some later time. This is known as an asynchronous callback, and is a common method of notifying a caller when a long process has completed. When a delegate is used in this fashion, the code using the delegate does not need any knowledge of the implementation of the method being used. The functionality is similar to the encapsulation interfaces provide.


Another common use of callbacks is defining a custom comparison method and passing that delegate to a sort method. It allows the caller's code to become part of the sort algorithm. The following example method uses the `Del` type as a parameter:

C#	
<pre>public static void MethodWithCallback(int param1, int param2, Del callback) { callback("The number is: " + (param1 + param2).ToString()); }</pre>	

You can then pass the delegate created above to that method:

C#	
<pre>MethodWithCallback(1, 2, handler);</pre>	


and receive the following output to the console:

console	
<pre>The number is: 3</pre>	

Using the delegate as an abstraction, `MethodWithCallback` does not need to call the console directly—it does not have to be designed with a console in mind. What `MethodWithCallback` does is simply prepare a string and pass the string to another method. This is especially powerful since a delegated method can use any number of parameters.


When a delegate is constructed to wrap an instance method, the delegate references both the instance and the method. A delegate has no knowledge of the instance type aside from the method it wraps, so a delegate can refer to any type of object as long as there is a method on that object that matches the delegate signature. When a delegate

is constructed to wrap a static method, it only references the method. Consider the following declarations:


C#	
<pre>public class MethodClass { public void Method1(string message) { } public void Method2(string message) { } }</pre>	

Along with the static `DelegateMethod` shown previously, we now have three methods that can be wrapped by a `Del` instance.

A delegate can call more than one method when invoked. This is referred to as multicasting. To add an extra method to the delegate's list of methods—the invocation list—simply requires adding two delegates using the addition or addition assignment operators ('+' or '+='). For example:

C#	
<pre>var obj = new MethodClass(); Del d1 = obj.Method1; Del d2 = obj.Method2; Del d3 = DelegateMethod; //Both types of assignment are valid. Del allMethodsDelegate = d1 + d2; allMethodsDelegate += d3;</pre>	

At this point `allMethodsDelegate` contains three methods in its invocation list—`Method1`, `Method2`, and `DelegateMethod`. The original three delegates, `d1`, `d2`, and `d3`, remain unchanged. When `allMethodsDelegate` is invoked, all three methods are called in order. If the delegate uses reference parameters, the reference is passed sequentially to each of the three methods in turn, and any changes by one method are visible to the next method. When any of the methods throws an exception that is not caught within the method, that exception is passed to the caller of the delegate and no subsequent methods in the invocation list are called. If the delegate has a return value and/or out parameters, it returns the return value and parameters of the last method invoked. To remove a method from the invocation list, use the [subtraction or subtraction assignment operators](#) (- or -=). For example:

C#	
<pre>//remove Method1 allMethodsDelegate -= d1;</pre>	

```
// copy AllMethodsDelegate while removing d2
Del oneMethodDelegate = allMethodsDelegate - d2;
```

Because delegate types are derived from `System.Delegate`, the methods and properties defined by that class can be called on the delegate. For example, to find the number of methods in a delegate's invocation list, you may write:

C#

 Copy

```
int invocationCount = d1.GetInvocationList().GetLength(0);
```

Delegates with more than one method in their invocation list derive from [MulticastDelegate](#), which is a subclass of `System.Delegate`. The above code works in either case because both classes support `GetInvocationList`.

Multicast delegates are used extensively in event handling. Event source objects send event notifications to recipient objects that have registered to receive that event. To register for an event, the recipient creates a method designed to handle the event, then creates a delegate for that method and passes the delegate to the event source. The source calls the delegate when the event occurs. The delegate then calls the event handling method on the recipient, delivering the event data. The delegate type for a given event is defined by the event source. For more, see [Events](#).

Comparing delegates of two different types assigned at compile-time will result in a compilation error. If the delegate instances are statically of the type `System.Delegate`, then the comparison is allowed, but will return false at run time. For example:

C#

 Copy

```
delegate void Delegate1();
delegate void Delegate2();

static void method(Delegate1 d, Delegate2 e, System.Delegate f)
{
    // Compile-time error.
    //Console.WriteLine(d == e);

    // OK at compile-time. False if the run-time type of f
    // is not the same as that of d.
    Console.WriteLine(d == f);
}
```

See also

- [C# Programming Guide](#)
- [Delegates](#)
- [Using Variance in Delegates](#)
- [Variance in Delegates](#)
- [Using Variance for Func and Action Generic Delegates](#)
- [Events](#)

Is this page helpful?

 Yes  No
