Contents

Delegates (C# Programming Guide)

5/4/2018 • 2 minutes to read • Edit Online

A delegate is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance.

Delegates are used to pass methods as arguments to other methods. Event handlers are nothing more than methods that are invoked through delegates. You create a custom method, and a class such as a windows control can call your method when a certain event occurs. The following example shows a delegate declaration:

```
public delegate int PerformCalculation(int x, int y);
```

Any method from any accessible class or struct that matches the delegate type can be assigned to the delegate. The method can be either static or an instance method. This makes it possible to programmatically change method calls, and also plug new code into existing classes.

> **NOTE**
> In the context of method overloading, the signature of a method does not include the return value. But in the context of delegates, the signature does include the return value. In other words, a method must have the same return type as the delegate.

This ability to refer to a method as a parameter makes delegates ideal for defining callback methods. For example, a reference to a method that compares two objects could be passed as an argument to a sort algorithm. Because the comparison code is in a separate procedure, the sort algorithm can be written in a more general way.

Delegates Overview

Delegates have the following properties:

- Delegates are like C++ function pointers but are type safe.
- Delegates allow methods to be passed as parameters.
- Delegates can be used to define callback methods.
- Delegates can be chained together; for example, multiple methods can be called on a single event.
- Methods do not have to match the delegate type exactly. For more information, see Using Variance in Delegates.
- C# version 2.0 introduced the concept of Anonymous Methods, which allow code blocks to be passed as parameters in place of a separately defined method. C# 3.0 introduced lambda expressions as a more concise way of writing inline code blocks. Both anonymous methods and lambda expressions (in certain contexts) are compiled to delegate types. Together, these features are now known as anonymous functions. For more information about lambda expressions, see Anonymous Functions.

In This Section

- Using Delegates

- When to Use Delegates Instead of Interfaces (C# Programming Guide)
- Delegates with Named vs. Anonymous Methods
- Anonymous Methods
- Using Variance in Delegates
- How to: Combine Delegates (Multicast Delegates)
- How to: Declare, Instantiate, and Use a Delegate

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

Featured Book Chapters

Delegates, Events, and Lambda Expressions in C# 3.0 Cookbook, Third Edition: More than 250 solutions for C# 3.0 programmers

Delegates and Events in Learning C# 3.0: Master the fundamentals of C# 3.0

See Also

Delegate
C# Programming Guide
Events

Using Delegates (C# Programming Guide)

5/4/2018 • 5 minutes to read • Edit Online

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:

```
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 Method. 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:

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

```
// 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:

```
public void MethodWithCallback(int param1, int param2, Del callback)
{
    callback("The number is: " + (param1 + param2).ToString());
}
```

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

```
MethodWithCallback(1, 2, handler);
```

and receive the following output to the console:

```
The number is: 3
```

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:

```
public class MethodClass
{
    public void Method1(string message) { }
    public void Method2(string message) { }
}
```

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:

```
MethodClass 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;
```

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 decrement or decrement assignment operator ('-' or '-='). For example:

```
//remove Method1
allMethodsDelegate -= d1;

// 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:

```
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:

```
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.
    System.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

Delegates with Named vs. Anonymous Methods (C# Programming Guide)

5/4/2018 • 2 minutes to read • Edit Online

A delegate can be associated with a named method. When you instantiate a delegate by using a named method, the method is passed as a parameter, for example:

```
// Declare a delegate:
delegate void Del(int x);

// Define a named method:
void DoWork(int k) { /* ... */ }

// Instantiate the delegate using the method as a parameter:
Del d = obj.DoWork;
```

This is called using a named method. Delegates constructed with a named method can encapsulate either a static method or an instance method. Named methods are the only way to instantiate a delegate in earlier versions of C#. However, in a situation where creating a new method is unwanted overhead, C# enables you to instantiate a delegate and immediately specify a code block that the delegate will process when it is called. The block can contain either a lambda expression or an anonymous method. For more information, see Anonymous Functions.

Remarks

The method that you pass as a delegate parameter must have the same signature as the delegate declaration.

A delegate instance may encapsulate either static or instance method.

Although the delegate can use an out parameter, we do not recommend its use with multicast event delegates because you cannot know which delegate will be called.

Example 1

The following is a simple example of declaring and using a delegate. Notice that both the delegate, `Del`, and the associated method, `MultiplyNumbers`, have the same signature

```
// Declare a delegate
delegate void Del(int i, double j);

class MathClass
{
    static void Main()
    {
        MathClass m = new MathClass();

        // Delegate instantiation using "MultiplyNumbers"
        Del d = m.MultiplyNumbers;

        // Invoke the delegate object.
        System.Console.WriteLine("Invoking the delegate using 'MultiplyNumbers':");
        for (int i = 1; i <= 5; i++)
        {
            d(i, 2);
        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }

    // Declare the associated method.
    void MultiplyNumbers(int m, double n)
    {
        System.Console.Write(m * n + " ");
    }
}
/* Output:
    Invoking the delegate using 'MultiplyNumbers':
    2 4 6 8 10
*/
```

Example 2

In the following example, one delegate is mapped to both static and instance methods and returns specific information from each.

```
// Declare a delegate
delegate void Del();

class SampleClass
{
    public void InstanceMethod()
    {
        System.Console.WriteLine("A message from the instance method.");
    }

    static public void StaticMethod()
    {
        System.Console.WriteLine("A message from the static method.");
    }
}

class TestSampleClass
{
    static void Main()
    {
        SampleClass sc = new SampleClass();

        // Map the delegate to the instance method:
        Del d = sc.InstanceMethod;
        d();

        // Map to the static method:
        d = SampleClass.StaticMethod;
        d();
    }
}
/* Output:
    A message from the instance method.
    A message from the static method.
*/
```

See Also

C# Programming Guide
Delegates
Anonymous Methods
How to: Combine Delegates (Multicast Delegates)
Events

How to: Combine Delegates (Multicast Delegates)(C# Programming Guide)

5/4/2018 • 2 minutes to read • Edit Online

This example demonstrates how to create multicast delegates. A useful property of delegate objects is that multiple objects can be assigned to one delegate instance by using the `+` operator. The multicast delegate contains a list of the assigned delegates. When the multicast delegate is called, it invokes the delegates in the list, in order. Only delegates of the same type can be combined.

The `-` operator can be used to remove a component delegate from a multicast delegate.

Example

```
using System;

// Define a custom delegate that has a string parameter and returns void.
delegate void CustomDel(string s);

class TestClass
{
    // Define two methods that have the same signature as CustomDel.
    static void Hello(string s)
    {
        System.Console.WriteLine("  Hello, {0}!", s);
    }

    static void Goodbye(string s)
    {
        System.Console.WriteLine("  Goodbye, {0}!", s);
    }

    static void Main()
    {
        // Declare instances of the custom delegate.
        CustomDel hiDel, byeDel, multiDel, multiMinusHiDel;

        // In this example, you can omit the custom delegate if you
        // want to and use Action<string> instead.
        //Action<string> hiDel, byeDel, multiDel, multiMinusHiDel;

        // Create the delegate object hiDel that references the
        // method Hello.
        hiDel = Hello;

        // Create the delegate object byeDel that references the
        // method Goodbye.
        byeDel = Goodbye;

        // The two delegates, hiDel and byeDel, are combined to
        // form multiDel.
        multiDel = hiDel + byeDel;

        // Remove hiDel from the multicast delegate, leaving byeDel,
        // which calls only the method Goodbye.
        multiMinusHiDel = multiDel - hiDel;

        Console.WriteLine("Invoking delegate hiDel:");
        hiDel("A");
        Console.WriteLine("Invoking delegate byeDel:");
        byeDel("B");
        Console.WriteLine("Invoking delegate multiDel:");
        multiDel("C");
        Console.WriteLine("Invoking delegate multiMinusHiDel:");
        multiMinusHiDel("D");
    }
}
/* Output:
Invoking delegate hiDel:
  Hello, A!
Invoking delegate byeDel:
  Goodbye, B!
Invoking delegate multiDel:
  Hello, C!
  Goodbye, C!
Invoking delegate multiMinusHiDel:
  Goodbye, D!
*/
```

See Also

MulticastDelegate
C# Programming Guide
Events

How to: Declare, Instantiate, and Use a Delegate (C# Programming Guide)

5/4/2018 • 4 minutes to read • Edit Online

In C# 1.0 and later, delegates can be declared as shown in the following example.

```
// Declare a delegate.
delegate void Del(string str);

// Declare a method with the same signature as the delegate.
static void Notify(string name)
{
    Console.WriteLine("Notification received for: {0}", name);
}
```

```
// Create an instance of the delegate.
Del del1 = new Del(Notify);
```

C# 2.0 provides a simpler way to write the previous declaration, as shown in the following example.

```
// C# 2.0 provides a simpler way to declare an instance of Del.
Del del2 = Notify;
```

In C# 2.0 and later, it is also possible to use an anonymous method to declare and initialize a delegate, as shown in the following example.

```
// Instantiate Del by using an anonymous method.
Del del3 = delegate(string name)
    { Console.WriteLine("Notification received for: {0}", name); };
```

In C# 3.0 and later, delegates can also be declared and instantiated by using a lambda expression, as shown in the following example.

```
// Instantiate Del by using a lambda expression.
Del del4 = name =>  { Console.WriteLine("Notification received for: {0}", name); };
```

For more information, see Lambda Expressions.

The following example illustrates declaring, instantiating, and using a delegate. The `BookDB` class encapsulates a bookstore database that maintains a database of books. It exposes a method, `ProcessPaperbackBooks`, which finds all paperback books in the database and calls a delegate for each one. The `delegate` type that is used is named `ProcessBookDelegate`. The `Test` class uses this class to print the titles and average price of the paperback books.

The use of delegates promotes good separation of functionality between the bookstore database and the client code. The client code has no knowledge of how the books are stored or how the bookstore code finds paperback books. The bookstore code has no knowledge of what processing is performed on the paperback books after it finds them.

Example

```
// A set of classes for handling a bookstore:
namespace Bookstore
{
    using System.Collections;

    // Describes a book in the book list:
    public struct Book
    {
        public string Title;        // Title of the book.
        public string Author;       // Author of the book.
        public decimal Price;       // Price of the book.
        public bool Paperback;      // Is it paperback?

        public Book(string title, string author, decimal price, bool paperBack)
        {
            Title = title;
            Author = author;
            Price = price;
            Paperback = paperBack;
        }
    }

    // Declare a delegate type for processing a book:
    public delegate void ProcessBookDelegate(Book book);

    // Maintains a book database.
    public class BookDB
    {
        // List of all books in the database:
        ArrayList list = new ArrayList();

        // Add a book to the database:
        public void AddBook(string title, string author, decimal price, bool paperBack)
        {
            list.Add(new Book(title, author, price, paperBack));
        }

        // Call a passed-in delegate on each paperback book to process it:
        public void ProcessPaperbackBooks(ProcessBookDelegate processBook)
        {
            foreach (Book b in list)
            {
                if (b.Paperback)
                    // Calling the delegate:
                    processBook(b);
            }
        }
    }
}


// Using the Bookstore classes:
namespace BookTestClient
{
    using Bookstore;

    // Class to total and average prices of books:
    class PriceTotaller
    {
        int countBooks = 0;
        decimal priceBooks = 0.0m;

        internal void AddBookToTotal(Book book)
        {
            countBooks += 1;
            priceBooks += book.Price;
```

```
            priceBooks += book.Price;
        }

        internal decimal AveragePrice()
        {
            return priceBooks / countBooks;
        }
    }

    // Class to test the book database:
    class TestBookDB
    {
        // Print the title of the book.
        static void PrintTitle(Book b)
        {
            System.Console.WriteLine("   {0}", b.Title);
        }

        // Execution starts here.
        static void Main()
        {
            BookDB bookDB = new BookDB();

            // Initialize the database with some books:
            AddBooks(bookDB);

            // Print all the titles of paperbacks:
            System.Console.WriteLine("Paperback Book Titles:");

            // Create a new delegate object associated with the static
            // method Test.PrintTitle:
            bookDB.ProcessPaperbackBooks(PrintTitle);

            // Get the average price of a paperback by using
            // a PriceTotaller object:
            PriceTotaller totaller = new PriceTotaller();

            // Create a new delegate object associated with the nonstatic
            // method AddBookToTotal on the object totaller:
            bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);

            System.Console.WriteLine("Average Paperback Book Price: ${0:#.##}",
                    totaller.AveragePrice());
        }

        // Initialize the book database with some test books:
        static void AddBooks(BookDB bookDB)
        {
            bookDB.AddBook("The C Programming Language", "Brian W. Kernighan and Dennis M. Ritchie", 19.95m,
true);
            bookDB.AddBook("The Unicode Standard 2.0", "The Unicode Consortium", 39.95m, true);
            bookDB.AddBook("The MS-DOS Encyclopedia", "Ray Duncan", 129.95m, false);
            bookDB.AddBook("Dogbert's Clues for the Clueless", "Scott Adams", 12.00m, true);
        }
    }
}
/* Output:
Paperback Book Titles:
   The C Programming Language
   The Unicode Standard 2.0
   Dogbert's Clues for the Clueless
Average Paperback Book Price: $23.97
*/
```

Robust Programming

- Declaring a delegate.

The following statement declares a new delegate type.

```
public delegate void ProcessBookDelegate(Book book);
```

Each delegate type describes the number and types of the arguments, and the type of the return value of methods that it can encapsulate. Whenever a new set of argument types or return value type is needed, a new delegate type must be declared.

- Instantiating a delegate.

  After a delegate type has been declared, a delegate object must be created and associated with a particular method. In the previous example, you do this by passing the `PrintTitle` method to the `ProcessPaperbackBooks` method as in the following example:

  ```
  bookDB.ProcessPaperbackBooks(PrintTitle);
  ```

  This creates a new delegate object associated with the static method `Test.PrintTitle`. Similarly, the non-static method `AddBookToTotal` on the object `totaller` is passed as in the following example:

  ```
  bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);
  ```

  In both cases a new delegate object is passed to the `ProcessPaperbackBooks` method.

  After a delegate is created, the method it is associated with never changes; delegate objects are immutable.

- Calling a delegate.

  After a delegate object is created, the delegate object is typically passed to other code that will call the delegate. A delegate object is called by using the name of the delegate object, followed by the parenthesized arguments to be passed to the delegate. Following is an example of a delegate call:

  ```
  processBook(b);
  ```

  A delegate can be either called synchronously, as in this example, or asynchronously by using `BeginInvoke` and `EndInvoke` methods.

See Also

C# Programming Guide
Events
Delegates