Vidya Vrat Agarwal Feb 12, 2023 792.6K 0 34

Introduction

Delegates in C# provide a way to define and execute callbacks. Their flexibility allows you to define the exact signature of the callback, and that information becomes part of the delegate type itself. Delegates are type-safe, object-oriented, and secure. In this article, you will learn how to create and manipulate delegate types and C# events that streamline the process of working with delegate types.

Delegates in C#

A Delegate is an abstraction of one or more function pointers (as existed in C++; the explanation about this is out of the scope of this article). The .NET has implemented the concept of function pointers in the form of delegates. With delegates, you can treat a function as data. Delegates allow functions to be passed as parameters, returned from a function as a value, and stored in an array. Delegates have the following characteristics:

- Delegates are derived from the System.MulticastDelegate class.
- They have a signature and a return type. A function that is added to delegates must be compatible with this signature.
- Delegates can point to either static or instance methods.
- Once a delegate object has been created, it may invoke the methods it points to at runtime.
- Delegates can call methods synchronously and asynchronously.

A delegate contains a couple of useful fields. The first is a reference to an object, and the second is a method pointer. When invoking the delegate, the instance method is called on the contained reference. However, if the object reference is null, then the runtime understands this to mean that the method is static. Moreover, invoking a delegate syntactically is the exact same as calling a regular function. Therefore, delegates are perfect for implementing callbacks.

Why Do We Need Delegates

Historically, the Windows API frequently used C-style function pointers to create callback functions. Using a callback, programmers could configure one function to report back to another function in the application. So the objective of using a callback is to handle button-clicking, menu-selection, and mouse-moving activities. But the problem with this traditional approach is that the callback functions were not type-safe. In the .NET framework, callbacks are still possible using delegates with a more

- The parameters of the method.

A delegate is a solution for situations where you want to pass methods around to other methods. You are so accustomed to passing data to methods as parameters that the idea of passing methods as an argument instead of data might sound a little strange. However, there are cases where you have a method that does something, for instance, invoking another method. You do not know at compile time what this second method is. That information is available only at runtime; hence Delegates are the device to overcome such complications.

Define a Delegate in C#

A delegate can be defined as a delegate type. Its definition must be similar to the function signature. A delegate can be defined in a namespace and within a class. A delegate cannot be used as a data member of a class or local variable within a method. The prototype for defining a delegate type is as the following;

accessibility delegate return type delegatename(parameterlist);

Delegate declarations look almost exactly like abstract method declarations, and you replace the abstract keyword with the delegate keyword. The following is a valid delegate declaration:

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

When the C# compiler encounters this line, it defines a type derived from MulticastDelegate, which also implements a method named Invoke that has the same signature as the method described in the delegate declaration. For all practical purposes, that class looks like the following:

```
public class operation : System.MulticastDelegate
{
    public double Invoke(int x, int y);
    // Other code
}
```

As you can see using ILDASM.exe, the compiler-generated operation class defines three public methods as in the following:

ILDASM

Figure 1.1

Delegates Sample Program

```
using System;

namespace Delegates
{
    // Delegate Definition
    public delegate int operation(int x, int y);

    class Program
    {
        // Method that is passes as an Argument
        // It has same signature as Delegates
        static int Addition(int a, int b)
        {
            return a + b;
        }

        static void Main(string[] args)
        {
            // Delegate instantiation
            operation obj = new operation(Program.Addition);

            // output
            Console.WriteLine("Addition is={0}",obj(23,27));
            Console.ReadLine();
        }
    }
}
```

Here, we are defining the delegate as a delegate type. The important point to remember is that the signature of the function reference by the delegate must match the delegate signature as:

```
// Delegate Definition
public delegate int operation(int x, int y);
```

can permit any method to take two integers and return an integer ... if you want to insert the target methods to a given delegate object, pass in the name of th delegate constructor as:

```
// Delegate Instantiation
operation obj = new operation(Program.Addition);
```

At this point, you can invoke the member pointed to using a direct function invocation as:

```
Console.WriteLine("Addition is={0}",obj(23,27));
```

Finally, when the delegate is no longer required, set the delegate instance to null.

Recall that .NET delegates are type-safe. Therefore, if you attempt to pass a delegate a method that does not match the signature pattern, the .NET will report a compile-time error.

Array of Delegates

Creating an array of delegates is similar to declaring an array of any type. The following example has a couple of static methods to perform specific math-related operations. Then you use delegates to call these methods.

```

using System;

namespace Delegates
{
    public class Operation
    {
        public static void Add(int a, int b)
        {
            Console.WriteLine("Addition={0}",a + b);
        }

        public static void Multiple(int a, int b)
        {
            Console.WriteLine("Multiply={0}", a * b);
        }
    }

    class Program
    {
        delegate void DelOp(int x, int y);

```

```
            {
                // Delegate instantiation
                DelOp[] obj =
```

```
                    new DelOp(Operation.Add),
                    new DelOp(Operation.Multiple)
                };

                for (int i = 0; i < obj.Length; i++)
                {
                    obj[i](2, 5);
                    obj[i](8, 5);
                    obj[i](4, 6);
                }
                Console.ReadLine();
            }
        }
    }
```

In this code, you instantiate an array of Delop delegates. Each element of the array is initialized to refer to a different operation implemented by the operation class. Then, you loop through the array and apply each operation to three different values. After compiling this code, the output will be as follows;

Array of Delegates
Figure 1.2

Anonymous Methods

Anonymous methods, as their name implies, are nameless methods. They prevent the creation of separate methods, especially when the functionality can be done without a new method creation. Anonymous methods provide a cleaner and more convenient approach while coding.

Define an anonymous method with the delegate keyword and a nameless function body. This code assigns an anonymous method to the delegate. The anonymous method must not have a signature. The signature and return type is inferred from the delegate type. For example, if the delegate has three parameters and returns a double type, the anonymous method would also have the same signature.

```
using System;

```

```
    class Program
    {
        delegate void operation();

        static void Main(string[] args)
        {
            // Delegate instantiation
            operation obj = delegate
            {
                Console.WriteLine("Anonymous method");
            };
            obj();

            Console.ReadLine();
        }
    }
}
```

The anonymous methods reduce the complexity of code, especially where several events are defined. With the anonymous method, the code does not perform faster. The compiler still defines methods implicitly.

Multicast Delegate

So far, you have seen a single method invocation/call with delegates. If you want to invoke/call more than one method, you must make an explicit call through a delegate more than once. However, it is possible for a delegate to do that via multicast delegates.

A multicast delegate is similar to a virtual container where multiple functions reference a stored invocation list. It successively calls each method in FIFO (first in, first out) order. When you wish to add multiple methods to a delegate object, you use an overloaded += operator rather than a direct assignment.

```
using System;

namespace Delegates
{
    public class Operation
```

```
        {
            Console.WriteLine("Addition={0}", a + 10);
```

```
        public static void Square(int a)
        {
            Console.WriteLine("Multiple={0}",a * a);
        }
    }
    class Program
    {
        delegate void DelOp(int x);

        static void Main(string[] args)
        {
            // Delegate instantiation
            DelOp obj = Operation.Add;
            obj += Operation.Square;

            obj(2);
            obj(8);

            Console.ReadLine();
        }
    }
}
```

The code above combines two delegates that hold the functions Add() and Square(). Here the Add() method executes first and Square() later. This is the order in which the function pointers are added to the multicast delegates.

To remove function references from a multicast delegate, use the overloaded -= operator that allows a caller to remove a method from the delegate object invocation list dynamically.

```
// Delegate instantiation
DelOp obj = Operation.Add;
obj -= Operation.Square;
```

Here when the delegate is invoked, the Add() method is executed, but Square() is not because we unsubscribed it with the -= operator from a given runtime notification.

methods invoked by a delegate throws an exception, the complet be aborted. You
can avoid such a scenario by iterating the method invocation list. The Delegate class defines a

```
using System;

namespace Delegates
{
    public class Operation
    {
        public static void One()
        {
            Console.WriteLine("one display");
            throw new Exception("Error");
        }
        public static void Two()
        {
            Console.WriteLine("Two display");
        }
    }

    class Program
    {
        delegate void DelOp();

        static void Main(string[] args)
        {
            // Delegate instantiation
            DelOp obj = Operation.One;
            obj += Operation.Two;

            Delegate[] del = obj.GetInvocationList();

            foreach (DelOp d in del)
            {
                try
                {
                    d();
                }
```

```
                    Console.WriteLine("Error ca
                }
```

```
                Console.ReadLine();
            }
        }
    }
```

When you run the application, you will see that the iteration continues with the next method even after an exception is caught, as in the following.

Multicast Delegate
Figure 1.3

Events

The applications and windows communicate via predefined messages. These messages contain various information to determine window and application actions. The .NET considers these messages as an event. You will handle the corresponding event if you need to react to a specific incoming message. For instance, when a button is clicked on a form, Windows sends a WM_MOUSECLICK message to the button message handler.

You don't need to build custom methods to add or remove methods to a delegate invocation list. C# provides the event keyword. When the compiler processes the event keyword, you can subscribe and unsubscribe methods and any necessary member variables for your delegate types. The syntax for the event definition should be as in the following:

Accessibility event delegatename eventname;

Defining an event is a two-step process. First, you need to define a delegate type that will hold the list of methods to be called when the event is fired. Next, you declare an event using the event keyword. To illustrate the event, we are creating a console application. In this iteration, we will define an event to add that is associated with a single delegate DelEventHandler.

```
using System;

namespace Delegates
{
    public delegate void DelEventHandler();

```

```
        public static event DelEventHandler ad
```

```
        {
            add += new DelEventHandler(USA);
            add += new DelEventHandler(India);
            add += new DelEventHandler(England);
            add.Invoke();

            Console.ReadLine();
        }
        static void USA()
        {
            Console.WriteLine("USA");
        }

        static void India()
        {
            Console.WriteLine("India");
        }

        static void England()
        {
            Console.WriteLine("England");
        }
    }
}
```

In the main method, we associate the event with its corresponding event handler with a function reference. Here, we fill the delegate invocation lists with a couple of defined methods using the +=operator. Finally, we invoke the event via the Invoke method.

Note: Event Handlers can't return a value. They are always void.

Let's use another example to get a better understanding of events. Here we are defining an event name as xyz and a delegate EventHandler with a string argument signature in the operation class. We are putting the implementation in the action method to confirm whether an event is fired or not.

```
namespace Delegates
{
```

```
    public class Operation
    {
        public event EventHandler xyz;

        public void Action(string a)
        {
            if (xyz != null)
            {
                xyz(a);
                Console.WriteLine(a);
            }
            else
            {
                Console.WriteLine("Not Registered");
            }
        }
    }

    class Program
    {
        public static void CatchEvent(string s)
        {
            Console.WriteLine("Method Calling");
        }

        static void Main(string[] args)
        {
            Operation o = new Operation();

            o.Action("Event Calling");
            o.xyz += new EventHandler(CatchEvent);

            Console.ReadLine();
        }
```

in the main method.

Finally, we are elaborating on the events with a realistic example of creating a custom button control over the Windows form that would be called from a console application. First, we inherit the program class from the Form class and define its associated namespace at the top. Then, we craft a custom button control in the program class constructor.

```

using System;
using System.Drawing;
using System.Windows.Forms;

namespace Delegates
{
    //custom delegate
    public delegate void DelEventHandler();

    class Program :Form
    {
        //custom event
        public event DelEventHandler add;

        public Program()
        {
            // desing a button over form
            Button btn = new Button();
            btn.Parent = this;
            btn.Text = "Hit Me";
            btn.Location = new Point(100,100);

            //Event handler is assigned to
            // the button click event
            btn.Click += new EventHandler(onClcik);
            add += new DelEventHandler(Initiate);

            //invoke the event
        //call when event is fired
        public void Initiate()
```

```
            Console.WriteLine("Event Initiated");
        }

        //call when button clicked
        public void onClcik(object sender, EventArgs e)
        {
            MessageBox.Show("You clicked me");
        }
        static void Main(string[] args)
        {
            Application.Run(new Program());

            Console.ReadLine();
        }
    }

}
```

We call the Windows Form using the Run method of the Application class. Finally, when the user runs this application, the Event fired message will be displayed on the screen, and a Windows Form with the custom button control will be displayed. When the button is clicked, a message box will be shown as in the following;

Figure4.jpg
Figure 1.4

Summary

This article taught us about delegates and events in C# and .NET.

Delegates in C# | Events | Events C# .NET | Multicast Delegate ecome a member C# Corner Login

Ask Question

SIMILAR ARTICLES

Learn

Learn C# 8.0

CHALLENGE YOURSELF

C# Skill

GET CERTIFIED

Java Developer

About Us Contact Us Privacy Policy Terms Media Kit Sitemap Report a Bug FAQ Partners

C# Tutorials Common Interview Questions Stories Consultants Ideas Certifications