Pass Method as Parameter using C#

617

183

I have several methods all with the same signature (parameters and return values) but different names and the internals of the methods are different. I want to pass the name of the method to run to another method that will invoke the passed in method.

```
public int Method1(string)
{
    ... do something
    return myInt;
}

public int Method2(string)
{
    ... do something different
    return myInt;
}

public bool RunTheMethod([Method Name passed in here] myMethodName)
{
    ... do stuff
    int i = myMethodName("My String");
    ... do more stuff
    return true;
}

public bool Test()
{
    return RunTheMethod(Method1);
}
```

This code does not work but this is what I am trying to do. What I don't understand is how to write the RunTheMethod code since I need to define the parameter.

c# .net methods delegates

9 Why don't you pass a delegate instead of the name of the method? – Mark Byers Jan 17 '10 at 21:02

10 Answers

766

✓

You can use the Func delegate in .net 3.5 as the parameter in your RunTheMethod method. The Func delegate allows you to specify a method that takes a number of parameters of a specific type and returns a single argument of a specific type. Here is an example that should work:

```
public class Class1
{
    public int Method1(string input)
    {
        //... do something
        return 0;
    }

    public int Method2(string input)
    {
        //... do something different
        return 1;
    }

    public bool RunTheMethod(Func<string, int> myMethodName)
    {
        //... do stuff
        int i = myMethodName("My String");
        //... do more stuff
        return true;
    }

    public bool Test()
    {
        return RunTheMethod(Method1);
    }
}
```

40 How would the Func call change if the Method has as signature of returning void and no parameters? I can't seem to get the syntax to work. – user31673 Jan 17 '10 at 21:38

186 @unknown: In that case it would be `Action` instead of `Func<string, int>`. – Jon Skeet Jan 17 '10 at 21:45

11 but now what if you want to pass in arguments to the method?? – john ktejik Feb 24 '14 at 2:28

34 @user396483 For example, `Action<int,string>` corresponds to a method which is taking 2 parameters (int and string) and returning void. – serdar Jun 12 '14 at 10:30

21 @NoelWidmer Using `Func<double,string,int>` corresponds to a method which takes 2 parameters (`double` and `string`) and returning `int`. Last specified type is the return type. You can use this delegate for up to 16 parameters. If you somehow need more, write your own delegate as `public delegate TResult Func<in T1, in T2, (as many arguments as you want), in Tn, out TResult>(T1 arg1, T2 arg2, ..., Tn argn);`. Please correct me if I misunderstood. – serdar Sep 11 '14 at 6:43

340

You need to use a *delegate*. In this case all your methods take a `string` parameter and return an `int` - this is most simply represented by the `Func<string, int>` delegate[1]. So your code can become correct with as simple a change as this:

```
public bool RunTheMethod(Func<string, int> myMethodName)
{
    // ... do stuff
    int i = myMethodName("My String");
    // ... do more stuff
    return true;
}
```

Delegates have a lot more power than this, admittedly. For example, with C# you can create a delegate from a *lambda expression*, so you could invoke your method this way:

```
RunTheMethod(x => x.Length);
```

That will create an anonymous function like this:

```
// The <> in the name make it "unspeakable" - you can't refer to this method directly
// in your own code.
```

and then pass that delegate to the `RunTheMethod` method.

You can use delegates for event subscriptions, asynchronous execution, callbacks - all kinds of things. It's well worth reading up on them, particularly if you want to use LINQ. I have an article which is *mostly* about the differences between delegates and events, but you may find it useful anyway.

[1] This is just based on the generic `Func<T, TResult>` delegate type in the framework; you could easily declare your own:

```
public delegate int MyDelegateType(string value)
```

and then make the parameter be of type `MyDelegateType` instead.

edited Sep 26 '16 at 5:31

answered Jan 17 '10 at 21:02
Jon Skeet
1119k 709 8142 8548

53 +1 This really is an amazing answer to rattle off in two minutes. – David Hall Jan 17 '10 at 21:09

3 While you can pass the function using delegates, a more traditional OO approach would be to use the strategy pattern. – Paolo Jan 17 '10 at 21:14

19 @Paolo: Delegates are just a very convenient implementation of the strategy pattern where the strategy in question only requires a single method. It's not like this is going *against* the strategy pattern - but it's a heck of a lot more convenient than implementing the pattern using interfaces. – Jon Skeet Jan 17 '10 at 21:18

5 Are the "classic" delegates (as known from .NET 1/2) still useful, or are they completely obsolete because of Func/Action? Also, isn't there a delegate keyword missing in your example `public **delegate** int MyDelegateType(string value)` ? – M4N Jan 17 '10 at 23:28

1 @Martin: Doh! Thanks for the fix. Edited. As for declaring your own delegates - it can be useful to give the type name some meaning, but I've rarely created my own delegate type since .NET 3.5. – Jon Skeet Jan 17 '10 at 23:50

From OP's example:

```
  public static int Method2(string mystring)
  {
      return 2;
  }
```

You can try Action Delegate! And then call your method using

```
  public bool RunTheMethod(Action myMethodName)
  {
       myMethodName();   // note: the return value got discarded
       return true;
  }

 RunTheMethod(() => Method1("MyString1"));
```

Or

```
 public static object InvokeMethod(Delegate method, params object[] args)
 {
      return method.DynamicInvoke(args);
 }
```

Then simply call method

```
 Console.WriteLine(InvokeMethod(new Func<string,int>(Method1), "MyString1"));

 Console.WriteLine(InvokeMethod(new Func<string, int>(Method2), "MyString2"));
```

edited Feb 25 at 21:38
themefield
785 11 16

answered Oct 14 '11 at 10:40
Zain Ali
10.6k 12 83 95

3 Thanks, this got me where I wanted to go since I wanted a more generic "RunTheMethod" method that would allow for multiple parameters. Btw your first `InvokeMethod` lambda call should be `RunTheMethod` instead – John Jan 27 '12 at 18:26

1 Like John, this really helped me have a move generic method. Thanks! – ean5533 Nov 6 '12 at 23:39

Join Stack Overflow to learn, share knowledge, and build your career.

Sign up with email Sign up with Google Sign up with Facebook ✕

If you want to pass parameters be aware of this: stackoverflow.com/a/5414539/2736039 – Ultimo_m Jun 12 '18 at 16:50

29

```
public static T Runner<T>(Func<T> funcToRun)
{
    //Do stuff before running function as normal
    return funcToRun();
}
```

Usage:

```
var ReturnValue = Runner(() => GetUser(99));
```

answered Aug 14 '14 at 2:50
kravits88
7,805 1 39 47

6 It's very userfully. With this way, can use one or many parameters. I guess, the most updated answer is this. – bafsar Jan 23 '15 at 16:53

I'd like to add one thing about this implementation. If the method you are going to pass has return type of void, then you can't use this solution. – Imants Volkovs May 19 '16 at 21:00

@ImantsVolkovs I believe you might be able to to modify this to use an Action instead of a Func, and change the signature to void. Not 100% sure though. – Shockwave Mar 27 '17 at 9:37

2 Is there any way to get the parameters passed to the function that is called? – Jimmy Aug 3 '17 at 20:43

11

You should use a `Func<string, int>` delegate, that represents a function taking a `string` as argument and returning an `int` :

```
public bool RunTheMethod(Func<string, int> myMethod) {
    // do stuff
    myMethod.Invoke("My String");
    // do stuff
```

Join Stack Overflow to learn, share knowledge, and build your career.

Sign up with email Sign up with Google Sign up with Facebook

```
public bool Test() {
    return RunTheMethod(Method1);
}
```

edited Aug 23 '13 at 21:37
daniel1426
127 1 13

answered Jan 17 '10 at 21:03
Bruno Reis
28.9k 10 98 138

3 This won't compile. The `Test` method should be `return RunTheMethod(Method1);` – p.s.w.g Aug 23 '13 at 21:34

6

If you want the ability to change which method is called at run time I would recommend using a delegate: http://www.codeproject.com/KB/cs/delegates_step1.aspx

It will allow you to create an object to store the method to call and you can pass that to your other methods when it's needed.

answered Jan 17 '10 at 21:05
MadcapLaugher
508 2 8

6

For sharing a as complete as possible solution, I'm going to end up with presenting three different ways of doing, but now I'm going to start from the most basic principle.

Brief introduction

All CLR (*Common Language Runtime*) languages (such as C# and Visual Basic) work under a VM called CLI (*Common Language Interpreter*) which runs the code on a higher level than native languages like C and C++ (which directly compile to machine code). It follows that methods aren't any kind of compiled block, but they are just structured elements which CLR recognize and use to pull out their body and rewing it to the inline instructions of the machine code. Thus, you cannot think to pass a method as a parameter, because a method doesn't produce any value by itself: it's not a valid expression! So that, you are going to stumble the delegate concept.

A delegate represents a pointer to a method. Because of (as I said above) a method is not a value, there's a special class in CLR languages: `Delegate` . That class wraps any method and you can implicitly cast any method to that.

Look at the following usage example:

```
static void MyMethod()
{
    Console.WriteLine("I was called by the Delegate special class!");
}

static void CallAnyMethod(Delegate yourMethod)
{
    yourMethod.DynamicInvoke(new object[] { /*Array of arguments to pass*/ });
}

static void Main()
{
    CallAnyMethod(MyMethod);
}
```

The three ways:

- **Way 1**
  Use the `Delegate` special class directly as the example above. The problem of this solution is that your code will be unchecked as you pass dynamically your arguments without restricting them to the types of those in the method declaration.
- **Way 2/3** Besides the `Delegate` special class, the delegates concept spreads to custom delegates, which are declarations of methods preceeded by the `delegate` keyword and they behave like a normal method. They are so checked, and you'll come up to a "*perfect*" code.

Look at the following example:

```
delegate void PrintDelegate(string prompt);

static void PrintSomewhere(PrintDelegate print, string prompt)
{
    print(prompt);
}

static void PrintOnScreen(string prompt)
{
    MessageBox.Show(prompt);
}

static void Main()
{
    PrintSomewhere(PrintOnConsole, "Press a key to get a message");
    Console.Read();
    PrintSomewhere(PrintOnScreen, "Hello world");
}
```

A second option by this way not to write your own custom delegate is using one of them declared within system libraries:

- `Action` wraps a `void` with no arguments.
- `Action<T1>` wraps a `void` with one argument.
- `Action<T1, T2>` wraps a `void` with two arguments.
- And so on...
- `Func<TR>` wraps a function with `TR` return type and with no arguments.
- `Func<T1, TR>` wraps a function with `TR` return type and with one argument.
- `Func<T1, T2, TR>` wraps a function with `TR` return type and with two arguments.
- And so on...

(This latter solution is that many people posted.)

edited May 13 at 21:46

answered May 15 '18 at 14:30
Davide Cannizzo
974 1 10 20

Shouldn't the return type of a Func<T> be the last? `Func<T1,T2,TR>` – hexterminator May 13 at 19:20

Oh, you're right. Just corrected – Davide Cannizzo May 13 at 21:45

2

I ended up here after doing my own searching for a solution to a similar question. I am building a plugin driven framework, and as part of it I wanted people to be able to add menu items to the applications menu to a generic list without exposing an actual `Menu` object because the framework may deploy on other platforms that don't have `Menu` UI objects. Adding general info about the menu is easy enough, but allowing the plugin developer enough liberty to create the callback for when the menu is clicked was proving to be a pain. Until it dawned on me that I was trying to re-invent the wheel and normal menus call and trigger the callback from events!

So the solution, as simple as it sounds once you realize it, eluded me until now.

Just create separate classes for each of your current methods, inherited from a base if you must, and just add an event handler to each.

answered Mar 29 '15 at 13:48
Wobbles
1,994 15 35

1

Here is an example Which can help you better to understand how to pass a function as a parameter.

Suppose you have *Parent* page and you want to open a child popup window. In the parent page there is a textbox that should be filled basing on child popup textbox.

Here you need to create a delegate.

Parent.cs // declaration of delegates public delegate void FillName(String FirstName);

Now create a function which will fill your textbox and function should map delegates

```
//parameters
public void Getname(String ThisName)
{
     txtname.Text=ThisName;
}
```

Now on button click you need to open a Child popup window.

```
  private void button1_Click(object sender, RoutedEventArgs e)
```

```
    }
```

IN ChildPopUp constructor you need to create parameter of 'delegate type' of parent //page

ChildPopUp.cs

```
    public  Parent.FillName obj;
    public PopUp(Parent.FillName objTMP)//parameter as deligate type
    {
        obj = objTMP;
        InitializeComponent();
    }



   private void OKButton_Click(object sender, RoutedEventArgs e)
    {


        obj(txtFirstName.Text);
        // Getname() function will call automatically here
        this.DialogResult = true;
    }
```

edited May 25 '16 at 9:58
SteakOverflow
2,388 1 24 42

answered Sep 24 '13 at 11:50
Shrikant-Divyanet Solution
39 5

Edited but quality of this answer could still be improved. – SteakOverflow May 25 '16 at 9:28

Here is an example without a parameter: http://en.csharp-online.net/CSharp_FAQ:_How_call_a_method_using_a_name_string

with params: http://www.daniweb.com/forums/thread98148.html#

answered Jan 17 '10 at 21:04
Jeremy Samuel
685 5 15

Note that the name of the method isn't in a string - it's actually as a method group. Delegates are the best answer here, not reflection. – Jon Skeet Jan 17 '10 at 21:06

i guess i missed the point – Jeremy Samuel Jan 17 '10 at 21:08

@Lette: In the method invocation, the expression used as the argument is a *method group*; it's the name of a method which is known at compile-time, and the compiler can convert this into a delegate. This is very different from the situation where the name is only known at execution time. – Jon Skeet Jan 17 '10 at 21:44