


[Languages](#) » [C#](#) » [General](#)

 License: [The Code Project Open License \(CPOL\)](#)

C# 1.0, C# 2.0, C# 3.0.NET 1.1, .NET 2.0, .NET 3.0, .NET 3.5, Dev

C# Delegates, Anonymous Methods, and Lambda Expressions – O My!

 By [Josh Fischer](#)

 Version: **3 (See All)**

 Posted: **17 Dec 2009**

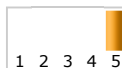
 Views: **1,773**

 Bookmarked: **27 times**

An explanation of the different ways to create delegates through a guided example that starts with .NET 1.1.

10 votes for this article.

Popularity: 5.00 Rating: 5.00 out of 5



Introduction

Delegates, anonymous methods, and lambda expressions can be very confusing in .NET. I think this is proven in code like the example below. Which one of the calls to `First` compiles? Which one returns the answer we are looking for; namely the Customer with the ID 5. The answer, incidentally, is that all six calls to `First` not only compile, but all find the correct customer and are functionally equivalent. If you are at all confused as to why this is true, this article is for you.

```
class Customer
{
    public int ID { get; set; }
    public static bool Test(Customer x)
    {
        return x.ID == 5;
    }
}
...
List<Customer> custs = new List<Customer>();
custs.Add(new Customer() { ID = 1 });
custs.Add(new Customer() { ID = 5 });

custs.First(new Func<Customer, bool>(delegate(Customer x) { return x.ID == 5; }));
custs.First(new Func<Customer, bool>((Customer x) => x.ID == 5));
custs.First(delegate(Customer x) { return x.ID == 5; });
custs.First((Customer x) => x.ID == 5);
custs.First(x => x.ID == 5);
custs.First(Customer.Test);
```

The setup – What are delegates?

Ok, so you've got a shopping cart class that processes a customer's order. Management has decided to give people discounts based on volume, price, etc. As part of this, they have implemented a factor that you must use when calculating an order. Ok, no big deal, you simply declare a variable to hold the 'magic discount' and proceed to code your algorithm.

```
class Program
{
    static void Main(string[] args)
    {
        new ShoppingCart().Process();
    }
}

class ShoppingCart
{
    public void Process()
    {
        int magicDiscount = 5;
        // ...
    }
}
```

Well, the next day, management, in their infinite wisdom, decides to change the discount amount based on the time of day; brilliant, I know. That's easy enough, however, so you simply make the changes in your code.

```
class ShoppingCart
{
    public void Process()
    {
        int magicDiscount = 5;
        if (DateTime.Now.Hour < 12)
        {
            magicDiscount = 10;
        }
    }
}
```

The following day, management once again changes things and adds even more logic (or il-logic) into the discount algorithm. 'That's enough', you say to yourself. How can I get this ridiculous algorithm out of my code and let someone else maintain the logic? What you want to do is hand over, or delegate, the responsibility to someone else. Fortunately, .NET has a mechanism to do this called, you guessed it, delegates.

Delegates

If you have a C/C++ background, the best way to describe delegates is to call them function pointers. For everyone else, think of them as a way to pass methods the same way you pass values and objects around. For example, the three lines below embody the same basic principle: you are passing, but not using, a piece of data to be used by the `Process` method.

```
// passing an integer value for the Process method to use
Process( 5 );
// passing a reference to an ArrayList object for the Process method to use
Process( new ArrayList() );
// passing a method reference for the Process method to call
Process( discountDelegate );
```

OK, so what is `discountDelegate` and how do I create one? How does the `Process` method use a delegate? The first thing we need to do is declare a delegate type in the same way we declare a class.

```
delegate int DiscountDelegate();
```

What this means is we now have a delegate type called `DiscountDelegate` that we can use in the same way we can use a class, struct, etc. It takes no parameters, but returns an integer. Just like a class, however, it isn't very useful until we create an instance of it. The trick to creating an instance of a delegate is to remember that a delegate is nothing more than a reference to a method. The key here is to realize that even though `DiscountDelegate` does not have any constructors, when creating one, there is an implicit constructor that wants a method matching its signature (no params, returning `int`). How do you 'give' the constructor a method? Well, .NET lets you simply type in the name in the same way you would call the method; all you do is omit the parentheses.

```
DiscountDelegate discount = new DiscountDelegate(class.method);
```

Before going further, let's go back to our example and put the pieces together. We will add a `Calculator` class to help us with the discount algorithm and give us some methods to point our delegate at.

```
delegate int DiscountDelegate();

class Program
{
    static void Main(string[] args)
    {
        Calculator calc = new Calculator();
        DiscountDelegate discount = null;
        if (DateTime.Now.Hour < 12)
        {
            discount = new DiscountDelegate(calc.Morning);
        }
    }
}
```

```

    }
    else if (DateTime.Now.Hour < 20)
    {
        discount = new DiscountDelegate(calc.Afternoon);
    }
    else
    {
        discount = new DiscountDelegate(calc.Night);
    }
    new ShoppingCart().Process(discount);
}
}

class Calculator
{
    public int Morning()
    {
        return 5;
    }
    public int Afternoon()
    {
        return 10;
    }
    public int Night()
    {
        return 15;
    }
}

class ShoppingCart
{
    public void Process(DiscountDelegate discount)
    {
        int magicDiscount = discount();
        // ...
    }
}

```

As you can see, we created a method in the `Calculator` class for each logical branch. We created an instance of `Calculator` and an instance of `DiscountDelegate` in the `Main` method that work together to set up the target method we want to call.

Great, now instead of having to worry about the logic in our `Process` method, we simply call the delegate we were given. Remember, we don't care how the delegate was created (or even when), we just call it like any other method when we need the value. As you can see, another way to think of a delegate is that it defers the execution of a method. The calculator method was chosen at some point in the past, but not actually executed until we called `discount()`. Looking at our solution, there still seems to be a lot of ugly code. Do we need a different method for every return value in the `Calculator` class? Of course, not; let's consolidate some of this mess.

```

delegate int DiscountDelegate();

class Program
{
    static void Main(string[] args)
    {
        new ShoppingCart().Process(new DiscountDelegate(Calculator.Calculate));
    }
}

class Calculator
{
    public static int Calculate()
    {
        int discount = 0;
        if (DateTime.Now.Hour < 12)
        {
            discount = 5;
        }
        else if (DateTime.Now.Hour < 20)
        {
            discount = 10;
        }
        else

```

```

        {
            discount = 15;
        }
        return discount;
    }
}

class ShoppingCart
{
    public void Process(DiscountDelegate discount)
    {
        int magicDiscount = discount();
        // ...
    }
}

```

There we go, much better. You'll notice we cleaned things up by making the `Calculate` method static and not bothering to keep a reference to the `DiscountDelegate` in the `Main` method. OK, so now, you know everything there is to know about delegates, right? Well, if this was 2004 and .NET 1.1, the answer would be 'yes', but fortunately, the framework has matured since then.

Lights, camera, action -or- We want the Func!

Generics were introduced in .NET 2.0, and Microsoft was nice enough to give us some common delegates to use so we wouldn't have to constantly define our own. These delegates are `Action` and `Func`, and the only difference between them is that `Func` matches methods that return a value and `Action` matches ones that do not.

This means that we do not need to declare our `DiscountDelegate` and instead can use `Func<int>` in its place. To demonstrate how the arguments work, let's assume management changes our algorithm once again, and now there is a special discount we need to account for. Easy enough, we will just ask for a boolean value in our `Calculate` method.

Our delegate's signature now becomes `Func<bool, int>`. Notice the `Calculate` method now takes a boolean parameter and that we call `discount()` using a boolean.

```

class Program
{
    static void Main(string[] args)
    {
        new ShoppingCart().Process(new Func<bool, int>(Calculator.Calculate));
    }
}

class Calculator
{
    public static int Calculate(bool special)
    {
        int discount = 0;
        if (DateTime.Now.Hour < 12)
        {
            discount = 5;
        }
        else if (DateTime.Now.Hour < 20)
        {
            discount = 10;
        }
        else if (special)
        {
            discount = 20;
        }
        else
        {
            discount = 15;
        }
        return discount;
    }
}

class ShoppingCart
{
    public void Process(Func<bool,int> discount)

```

```

    {
        int magicDiscount = discount(false);
        int magicDiscount2 = discount(true);
    }
}

```

Not bad, we saved another line of code, are we done now? Of course not, we can save even more time by using type inference. .NET allows us to completely omit the explicit creation of `Func<bool, int>` as long as the method we pass has the proper signature of the expected delegate.

```

// works because Process expects a method that takes a bool and returns int
new ShoppingCart().Process(Calculator.Calculate);

```

Up to this point, we have shaved our code down by first omitting the need for a custom delegate and then eliminating the need to even explicitly create a `Func` delegate. Is there anything else we can do to trim our line count? Well, since we are only half way through the article, the answer is obviously 'yes'.

Anonymous methods

Anonymous methods let you declare a method body without giving it a name. Behind the scenes, they exist as 'normal' methods; however, there is no way to explicitly call them in your code. Anonymous methods can only be created when using delegates and, in fact, they are created using the `delegate` keyword.

```

class Program
{
    static void Main(string[] args)
    {
        new ShoppingCart().Process(
            new Func<bool, int>(delegate(bool x) { return x ? 10 : 5; }
        ));
    }
}

```

As you can see, we completely removed the need for the `Calculator` class. You can put as much or as little logic between the curly braces as you would in any other method. If you are having trouble seeing how this could work, pretend the declaration `delegate(bool x)` is actually a method signature and not a keyword. Picture that piece of code inside a class. `delegate(bool x) { return 5; }` is a perfectly legitimate method declaration (yes, we would have to add the return type); it just so happens `delegate` is a reserved word, and in this case, it makes the method anonymous.

Well, I'm sure you know by now there is even more trimming we can do. Naturally, we can omit the need to explicitly declare the `Func` delegate; .NET takes care of that for us when we use the `delegate` keyword.

```

class Program
{
    static void Main(string[] args)
    {
        new ShoppingCart().Process(
            delegate(bool x) { return x ? 10 : 5; }
        );
    }
}

```

The true power of anonymous methods can be seen when using .NET methods that expect delegates as parameters and when responding to events. Previously, you had to create a method for every possible action you wanted to take. Now you can just create them inline and avoid polluting your namespace.

```

// creates an anonymous comparer
custs.Sort(delegate(Customer c1, Customer c2)
{
    return Comparer<int>.Default.Compare(c1.ID, c2.ID);
});

// creates an anonymous event handler
button1.Click += delegate(object o, EventArgs e)
    { MessageBox.Show("Click!"); };

```

Well, we have covered a lot of ground and we have streamlined our code quite a bit. This seems like a good place to stop and, in fact, it was until .NET 3.0 was released.

Lambda expressions

From MSDN: 'A lambda expression is an anonymous function that can contain expressions and statements, and can be used to create delegates or expression tree types.' OK, you should understand the 'used to create delegates' part, but what about this 'expression' stuff? Well, to be honest, expressions and expression trees are beyond the scope of this article. The only thing we need to understand about them right now is that an expression is code (yes, your C# code) represented as data and/or objects in a running .NET application. To quote the mighty Jon Skeet: 'Expression trees are a way of expressing logic so that other code can interrogate it. When a lambda expression is converted into an expression tree, the compiler doesn't emit the IL for the lambda expression; it emits IL which will build an expression tree representing the same logic.'

What we need to focus on is that lambda expressions replace anonymous methods and add many features. Looking back at our last example, we had trimmed the code down to where we were basically creating the entire discount algorithm in a single line.

```
class Program
{
    static void Main(string[] args)
    {
        new ShoppingCart().Process(
            delegate(bool x) { return x ? 10 : 5; }
        );
    }
}
```

Would you believe we can make this even shorter? Lambda expressions use the 'goes to' operator `=>` to indicate what parameters are being passed to the expression. The compiler takes this a step further, and allows us to omit the types and infers them for us. If you have two or more parameters, you need to use parentheses: `(x,y) =>`. If you only have one, however, you don't need them: `x =>`.

```
static void Main(string[] args)
{
    Func<bool, int> del = x => x ? 10 : 5;
    new ShoppingCart().Process(del);
}
// even shorter...
static void Main(string[] args)
{
    new ShoppingCart().Process(x => x ? 10 : 5);
}
```

Yep, that's it. The boolean type of `x` is inferred and so is the return type, because `Process` takes a `Func<bool, int>`. If we want to implement the full code block like we had before, all we need to do is add the brackets.

```
static void Main(string[] args)
{
    new ShoppingCart().Process(
        x => {
            int discount = 0;
            if (DateTime.Now.Hour < 12)
            {
                discount = 5;
            }
            else if (DateTime.Now.Hour < 20)
            {
                discount = 10;
            }
            else if(x)
            {
                discount = 20;
            }
            else
            {
                discount = 15;
            }
        }
    );
}
```

```

    }
    return discount;
  });
}

```

One last thing...

There is an important difference between using brackets and not using them. When you use them, you are creating a 'statement lambda', otherwise it is 'expression lambda'. Statement lambdas can execute multiple statements (hence the need for brackets) and can not create expression trees. You will probably only run into this problem when working with the `IQueryable` interface. The example below shows the problem.

```

List<string> list = new List<string>();
IQueryable<string> query = list.AsQueryable();
list.Add("one");
list.Add("two");
list.Add("three");

string foo = list.First(x => x.EndsWith("o"));
string bar = query.First(x => x.EndsWith("o"));
// foo and bar are now both 'two' as expected
foo = list.First(x => { return x.EndsWith("e"); }); //no error
bar = query.First(x => { return x.EndsWith("e"); }); //error
bar = query.First((Func<string,bool>)(x => { return x.EndsWith("e"); })); //no error

```

The second assignment of `bar` fails at compile time. This is because `IQueryable.First` expects an expression as a parameter whereas the extension method `List<T>.First` expects a delegate. You can force the lambda to evaluate to a delegate (and use the `First`'s method overload) by making a cast as I did in the third assignment to `bar`.

It's hard to end the discussion here, but I feel I must. Lambdas are basically broken into two categories; the kind that create anonymous methods and delegates, and the kind that create expressions. Expressions are an article by themselves and are not necessarily required knowledge to be a .NET developer (although their implementation in LINQ certainly is).

Conclusion

I hope this article has accomplished the goal of setting straight the six confusing delegate calls at the beginning of the article and explaining the interplay between delegates, anonymous methods, and lambda expressions.

History

- 12/17/2009: Initial version.


License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

About the Author

Josh Fischer



Occupation: Software Developer (Senior)
 Location:  United States

Member

Discussions and Feedback

 **2 messages** have been posted for this article. Visit <http://www.codeproject.com/KB/cs/DelegatesOMy.aspx> to post and view comments on this article, or click [here](#) to get a print view with messages.

[PermaLink](#) | [Privacy](#) | [Terms of Use](#)
Last Updated: 17 Dec 2009
Editor: [Smitha Vijayan](#)

Copyright 2009 by Josh Fischer
Everything else Copyright © [CodeProject](#), 1999-2009
Web22 | [Advertise on the Code Project](#)