The implementation of anonymous methods in C# and its consequences (part 1)

devblogs.microsoft.com/oldnewthing/20060802-00

August 2, 2006

Raymond Chen

You may not even have realized that there are two types of anonymous methods. I'll call them the easy kind and the hard kind, not because they're actually easy and hard for you the programmer, but because they are easy and hard for the compiler.

The easy kind is the anonymous method that doesn't use any local variables from its lexically-enclosing method. These are anonymous methods that could have been their own separate member functions; all the anonymization does is save you the trouble of coming up with names for them:

```
class MyClass1 {
 int v = 0;
 delegate void MyDelegate(string s);
 MyDelegate MemberFunc()
 {
  int i = 1;
  return delegate(string s) {
          System.Console.WriteLine(s);
         };
  }
}
```

This particular anonymous method doesn't access any `MyClass1` members, nor does it access the local variables of the `MemberFunc` function; therefore, it can be converted to a static method of the `MyClass1` class:

```
class MyClass1_converted {
 int v = 0;
 delegate void MyDelegate(string s);
 // Autogenerated by the compiler
 static void __AnonymousMethod$0(string s)
 {
  System.Console.WriteLine(s);
 }
 MyDelegate MemberFunc()
 {
  int i = 1;
  return __AnonymousMethod$0;
  // which is in turn shorthand for
  // return new MyDelegate(MyClass1.__AnonymousMethod$0);
  }
}
```

All the compiler did was give your anonymous methods a name and use that name in place of the " `delegate (...) { ... }` ". (Note that all compiler-generated names I use here are purely illustrative. The actual compiler-generated name will be something different.)

On the other hand, if your anonymous method used the `this` parameter, then that makes it an instance method instead of a static method:

```
class MyClass2 {
 int v = 0;
 delegate void MyDelegate(string s);
 MyDelegate MemberFunc()
 {
  int i = 1;
  return delegate(string s) {
          System.Console.WriteLine("{0} {1}", v, s);
         };
  }
}
```

The anonymous method in `MyClass2` uses the `this` keyword implicitly (to access the member variable `v`). Therefore, the conversion is to an instance member rather than to a static member.

```
class MyClass2_converted {
 int v = 0;
 delegate void MyDelegate(string s);
 // Autogenerated by the compiler
 void __AnonymousMethod$0(string s)
 {
  System.Console.WriteLine("{0} {1}", v, s);
 }
 MyDelegate MemberFunc()
 {
  int i = 1;
  return this.__AnonymousMethod$0;
  // which is in turn shorthand for
  // return new MyDelegate(this.__AnonymousMethod$0);
  }
}
```

So far, we've only dealt with the easy cases. The transformation is local and not particularly complicated. These are the sorts of transformations you could make yourself without too much difficulty in the absence of anonymous methods.

The hard case is where things get interesting. The body of an anonymous method is permitted to access the local variables of its lexically-enclosing method, in which case the compiler needs to keep those variables alive so that the body of your anonymous method can access them. Here's a sample anonymous method that accesses local variables from its lexically-enclosing method:

```
class MyClass3 {
 int v = 0;
 delegate void MyDelegate(string s);
 MyDelegate MemberFunc()
 {
  int i = 1;
  return delegate(string s) {
          System.Console.WriteLine("{0} {1} {2}", i++, v, s);
         };
  }
}
```

In this example, the anonymous method prints "1 v s" the first time it is called, then "2 v s" the second time it is called, and so on, with the integer increasing by one. (And where `v s` are the current values of `v` and `s`, of course.) This happens because the `i` variable that the anonymous method is accessing is the same one each time, and it's the same `i` that the `MemberFunc` method was using, too. If the function were rewritten as

```
class MyClass4 {
 int v = 0;
 delegate void MyDelegate(string s);
 MyDelegate MemberFunc()
 {
  int i = 0;
  MyDelegate d = delegate(string s) {
          System.Console.WriteLine("{0} {1} {2}", i++, v, s);
         };
  i = 1;
  return d;
  }
}
```

the behavior would be the same as in `MyClass3`. The creation of the delegate from the anonymous method does **not** make a copy of the `i` variable; changes to the `i` variable in the `MemberFunc` are visible to the anonymous method because both are accessing the **same** variable.

When faced with this “hard” type of anonymous method, wherein variables are shared with the lexically-enclosing method, the compiler generates a helper class:

```
class MyClass3_converted {
 int v = 0;
 delegate void MyDelegate(string s);
 // Autogenerated by the compiler
 class __AnonymousClass$0 {
  MyClass this$0;
  int i;
  public void __AnonymousMethod$0(string s)
  {
    System.Console.WriteLine("{0} {1} {2}", i++, this$0.v, s);
  }
 }
 MyDelegate MemberFunc()
 {
  __AnonymousClass$0 locals$ = new __AnonymousClass$0();
  locals$.this$0 = this;
  locals$.i = 0;
  return locals$.__AnonymousMethod$0;
  // which is in turn shorthand for
  // return new MyDelegate(locals$.__AnonymousMethod$0);
  }
}
```

Wow, there was a lot of rewriting this time. A helper class was created to contain the local variables that were shared between the `MemberFunc` function and the anonymous method (in this case, just the variable `i`), as well as the hidden `this` parameter (which I have called `this$`). In the `MemberFunc` function, access to that shared variable is done through this anonymous class, and the anonymous method that you wrote is an anonymous method on the anonymous class.

Notice that the assignment to `i` in `MemberFunc` modifies the copy inside `locals$`, which is the same object that the anonymous method will be using when it runs. That's why it prints "1 v s" the first time: The value had already been changed to 1 by the time the delegate ran for the first time.

Those who have done a good amount of C++ programming (or C# 1.0 programming) are well familiar with this technique, since C++ callbacks typically are given only one context variable; that context variable is usually a pointer to a larger structure that contains all the complex context you really want to operate on. C# 1.0 programmers went through a similar exercise. The "hard" type of anonymous method provides syntactic sugar that saves you the hassle of having to declare and manage the helper class.

If you thought about it some, you'd have realized that the way it's done is pretty much the only way it could have been done. It turns out that most computer programming doesn't consist of being clever or making hard decisions. You just have one kernel of an idea ("hey let's have anonymous methods") and then the rest is just doing what has to be done, no actual decisions needed. You just do the obvious thing. Most programming consists of just doing the obvious thing.

Okay, so that's a quick introduction to the implementation of anonymous methods in C#. Mind you, this information isn't just for your personal edification. It's actually important that you understand how these works (and not just treat it as "magic"), because lack of said understanding can lead to subtle programming errors. We'll look at those types of errors over the next few days.

Update: This behavior changed in Visual Studio 2015 with the switch to the Roslyn compiler. For performance reasons, anonymous methods are now always instance methods, even if they capture nothing.

Raymond Chen

Follow