The implementation of anonymous methods in C# and its consequences (part 3)

devblogs.microsoft.com/oldnewthing/20060804-00

August 4, 2006

Raymond Chen

Last time we saw how the implementation details of anonymous methods can make themselves visible when you start taking a delegate apart by looking at its `Target` and `Method`. This time, we'll see how an innocuous code change can result in disaster due to anonymous methods.

Occasionally, I see people arguing over where local variables should be declared. The "decentralists" believe that variables should be declared as close to their point of first use as possible:

```
void MyFunc1()
{
 ...
 for (int i = 0; i < 10; i++) {
  string s = i.ToString();
  ...
 }
 ...
}
```

On the other hand, the "consolidators" believe that local variables should be declared outside of loops.

```
void MyFunc2()
{
 ...
 string s;
 for (int i = 0; i < 10; i++) {
  s = i.ToString();
  ...
 }
 ...
}
```

The "consolidators" argue that hoisting the variable `s` means that the compiler only has to create the variable once, at function entry, rather than each time through the loop.

As a result, you can find yourself caught in a struggle between the "decentralists" and the "consolidators" as members of each school touch a piece of code and "fix" the local variable declarations to suit their style.

And then there are the "peacemakers" who step in and say, "Look, it doesn't matter. Can't we all just get along?"

While I admire the desire to have everyone get along, the claim that it doesn't matter is unfortunately not always true. Let's stick some nasty code in where the dots are:

```
delegate void MyDelegate();
void MyFunc1()
{
 MyDelegate d = null;
 for (int i = 0; i < 10; i++) {
  string s = i.ToString();
  d += delegate() {
   System.Console.WriteLine(s);
  };
 }
 d();
}
```

Since the `s` variable is declared inside the loop, each iteration of the loop gets its own copy of `s`, which means that each delegate **gets its own copy of `s`**. The first time through the loop, an `s` is created with the value `"0"` and that `s` is used by the first delegate. The second time through the loop, a new `s` is created with the value `"1"`, and that new `s` is used by the second delegate. The result of this code fragment is ten delegates, each of which prints a different number from 0 to 9.

Now, a "consolidator" looks at this code and says, "How inefficient, creating a new `s` each time through the loop. I shall hoist it and bask in the accolades of my countrymen."

```
delegate void MyDelegate();
void MyFunc2()
{
 MyDelegate d = null;
 string s;
 for (int i = 0; i < 10; i++) {
  s = i.ToString();
  d += delegate() {
   System.Console.WriteLine(s);
  };
 }
 d();
}
```

If you run this fragment, you get different behavior. A single `s` variable is created for all the loop iterations to share. The first time through the loop, the value of `s` is `"0"`, and then the first delegate is created. The second loop iteration changes the value of `s` to `"1"` before creating the second delegate. Repeat for the remaining eight delegates, and at the end of the loop, the value of `s` is `"9"`, and ten delegates have been added to `d`. When `d` is invoked, all the delegates print the value of the `s` variable, which they are sharing and which has the value `"9"`. The result: `9` is printed ten times.

Now, I happen to have constructed this scenario to make the "consolidators" look bad, but I could also have written it to make the "decentralists" look bad for pushing a variable declaration into a loop scope when it should have remained outside. (All you have to do is read the above scenario in reverse.)

The point of this little exercise is that when a "consolidator" or a "decentralist" goes through an entire program "tuning up" the declarations of local variables, the result can be a broken program, even though the person making the change was convinced that their change "had no effect; I was just making the code prettier / more efficient".

What's the conclusion here?

Write what you mean and mean what you write. If the precise scope of a variable is important, make sure to comment it as such so that somebody won't mess it up in a "clean-up" pass over your program. If there are two ways of writing the same thing, then write the one that is more maintainable. And if you feel that one method is superior from a performance point of view, then (1) make sure you're right, and (2) make sure it matters.

Update: In C# 5, the rules for the `foreach` statement changed in a way that affects lambda capture: The control variable of the `foreach` is now scoped to the loop body, which means that capturing it in a lambda captures the current iteration, because each iteration gets a separate copy of th3e variable. That doesn't affect our `for` loop above, but it is worth calling out.

Raymond Chen

Follow