

CODING IN THE SHADOWS

WEDNESDAY, JANUARY 7, 2009

Yield, and the C# state machine

For those of you who don't already know, the *C#* compiler has its own state machine generator that you can use. It's true - it's called the 'yield' statement!

Yield is used to create implementations of the Enumerable pattern - a software pattern that allows you to treat a collection of things as an enumeration, over which you can perform some process. In *C#*, you consume an enumeration via the 'foreach' statement, like so:

```
1 IEnumerable<string> ies = new List<string>() { "hush!", "tenth!", "lqwe!", "fgh!" };
2
3 foreach (var s in ies) {
4     Console.WriteLine(s);
5 }
```

Iterating over a List of strings

Before *C#* 2.0, creating a custom enumerable meant implementing the Enumerable Pattern via *IEnumerator* and *IEnumerable*. I don't feel like going through this, so here's a short and sweet example I found online.

This is a fairly common implementation. In fact, it is so common that when the *C#* language designers were conceiving 2.0 of their product, they chose to make it a first-class compiler-driven feature. Enter the yield keyword, which is capable of turning the example above into the following code:

```
1 public IEnumerable<char> MyChars() {
2     yield return 'A';
3     yield return 'B';
4     yield return 'C';
5     yield return 'D';
6 }
```

Implementing IEnumerable and IEnumerator via the yield keyword

This is much more straightforward, but there is some magic happening here that allows this to happen. First of all, the Enumerable pattern includes the requirement that the processing of the enumeration be lazy - that is, evaluated on a need basis. This allows an enumeration to contain an effectively infinite amount of items. Such an enumeration can be created using this code:

```
1 public IEnumerable<bool> infiniteAlternatingBools() {
2     bool cur = false;
3     while (true) {
4         yield return cur;
5         cur = !cur;
6     }
7 }
```

Creating an enumeration of an infinite pattern of alternating booleans

This code generates a list of alternating booleans - True / False / True / False - forever. Surely, this code will result in a locked-up process. Not so, fortunately for us, because behind the scenes (in the magic part) this code is expanded into a proper implementation of the Enumerable pattern - lazy evaluation included. Only when you request the next value is it generated and then provided, meaning this infinite generation can be short-circuited at any moment.

How? Magic, like I said - although this magic can be explained through gratuitous use of .NET Reflector. According to .NET Reflector, my infiniteAlternatingBools() method looks like this:

```
1 public IEnumerable<bool> infiniteAlternatingBools()
2 {
3     <infiniteAlternatingBools.d__5 d__ = new <infiniteAlternatingBools.d__5>(2);
4     d__.<4__this = this;
5     return d__;
6 }
```

Reflector output of infiniteAlternatingBools method

What? This is a mess. What is reflector telling me about this code?

In a nutshell, Reflector is saying that the *C#* compiler has, behind my back, taken the code I wrote and moved it into an anonymous private class. The constructor of that class takes an integer in its constructor, which the rewritten method initializes to -2. It also sets a 'this' property to the class that contains infiniteAlternatingBools - probably allowing it to access the original class's private members. Then the rewritten method returns the instance of that anonymous class - which suggests that it implements *IEnumerable<bool>*.

Kind of rude, don't you think? Replacing our carefully written infinite loop with some object creation? Actually the *C#* compiler has done us a favor - if you look in the anonymous class it generated you'll find the original code you wrote, albeit in a form you might not fully recognize. Here's the listing of the class (cleaned up a little bit from the Reflector version, which contains illegal characters):

```
1 [CompilerGenerated]
2 private sealed class d__5 :
3     IEnumerable<bool>, IEnumerable,
4     IEnumerable<bool>, IEnumerator,
5     IDisposable {
6
7     // Fields
8     private int state;
9     private bool current;
10    public Program.anon__this;
11    private int initialThreadId;
12    public bool _6;
13
14    // Methods
15    [DebuggerHidden]
16    public d__5(int state) {
17        this.state = state;
18        this.initialThreadId = Thread.CurrentThread.ManagedThreadId;
19    }
20
21    public bool MoveNext() {
22        switch (this.state) {
23            case 0:
24                this.state = -1;
25                this._6 = false;
26                break;
27
28            case 1:
29                this.state = -1;
30                this._6 = !this._6;
31                break;
32
33            default:
34                return false;
35        }
36        this.current = this._6;
37        this.state = 1;
38        return true;
39    }
40
41    [DebuggerHidden]
42    IEnumerator<bool> IEnumerable<bool>.GetEnumerator() {
43        if ((Thread.CurrentThread.ManagedThreadId == this.initialThreadId) && (this.state == -2)) {
44            this.state = 0;
45            return this;
46        }
47        Program.anon.d__5 d__ = new Program.anon.d__5(0);
48        d__._this = this;
49        return d__;
50    }
51
52    [DebuggerHidden]
53    IEnumerator IEnumerable.GetEnumerator() {
54        return this;
55    }
56
57    [DebuggerHidden]
58    void IEnumerator.Reset() {
59        throw new NotSupportedException();
60    }
61
62    void IDisposable.Dispose() {
63    }
64
65    // Properties
66    bool IEnumerator<bool>.Current {
67        [DebuggerHidden]
68        get {
69            return this.current;
70        }
71    }
72
73    object IEnumerator.Current {
74        [DebuggerHidden]
75        get {
76            return this.current;
77        }
78    }
79 }
```

Listing of the generated anonymous class implementing the Enumerable pattern

This listing is a bit hard to understand. There are fields called state, current, _this, initialThreadId, and _6. There are the *IEnumerable* and *IEnumerator* implementations - *MoveNext*, *Current*, *GetEnumerator*, and *Reset*. There's a constructor (taking an int). What can all this mean? More importantly, where's my infinite loop?

There is no infinite loop. My code is still here, but it's been turned into a state machine. The value that gets passed in to the constructor (-2) tells this state machine that it's in the initial state. When *GetEnumerator* is called, it checks to see if it's in its initial state - if it's not, it creates a new version of itself and returns that - but if it is, then it moves into state 0 and returns itself. When *MoveNext* is called, it uses the state to determine what value to set as the 'current' property. At state 0, the initial value is returned - which the *C#* compiler correctly determined to be false, given my initial 'bool cur = false;' statement. It also moves into state 1. Subsequent calls to *MoveNext* will call my code which alternates this _6 value, which is a boolean, between true and false - mimicking the behavior I coded.

My infinite loop turned into a lazily evaluated Enumerable Pattern implementation which uses a state machine to decide on what the 'current' value should be whenever *MoveNext* is called. pretty damned cool if you ask me.

The coolest thing about this is that you can use any *C#* constructs you want in your enumerable, and create some incredibly complex generators. In my case, I'm taking advantage of the built-in state machine to create a workflow-like process. One place I plan on using this is in EverHarvest 2. Here's a simplified version of what my workflow might look like once fully implemented:

```
1 public IEnumerable<WorkUnit> Workflow() {
2     yield return new Initialize();
3
4     while (true) {
5         var wp = GetNextWaypoint();
6         var wtw = new WalkingToWaypoint(wp);
7         while (!wtw.ReachedWaypoint) {
8             yield return wtw;
9
10            var tgt = new Targetting();
11            yield return tgt;
12
13            if (tgt.FoundTarget) {
14                if (!IsNode(tgt.TargetName)) {
15                    var wtn = new WalkingToHarvestable(tgt.TargetName, tgt.Target.Location);
16                    while (!wtn.ReachedNode) {
17                        yield return wtn;
18                    }
19
20                    var h = new Harvesting();
21                    while (!h.DoneHarvesting) {
22                        yield return h;
23                    }
24                }
25            }
26        }
27    }
28 }
```

EverHarvest Workflow example

Notice how simple this is to understand. It reads very procedurally, and yet because this is lazily evaluated, this process can be interrupted at any of the yield points. This control lies with the code that is enumerating through the workflow - that code can act as the gatekeeper, deciding when to get the next work item, when to execute it, when to break out of the loop, what data each bit should have, etc. This can be done in a foreach statement, or I can use the older *MoveNext* / *Current* members.

I hope this helps those of you who are still reading to understand how the yield statement can be used to take advantage of the state machine functionality that the *C#* compiler provides for us. In terms of readability and maintenance, it has proven to be a real boon for me. I hope this has helped you to find the same benefit.


Edit: Here's the disassembled *MoveNext()* method from Reflector - I haven't cleaned it up a bit. Lots of red squigglies in this one...

```
1 private bool MoveNext()
2 {
3     bool CS$4$0002;
4     switch ((this.<>1__state)
5     {
6         case 0:
7             this.<>1__state = -1;
8             this.<>2__current = new Initialize();
9             this.<>1__state = 1;
10            return true;
11
12            case 1:
13                this.<>1__state = -1;
14                goto Label_01CB;
15
16                case 2:
17                    goto Label_00B4;
18
19                    case 3:
20                        goto Label_00E0;
21
22                        case 4:
23                            goto Label_0159;
24
25                            case 5:
26                                goto Label_0198;
27
28                                default:
29                                    return false;
30        }
31    Label_01CB:
32    CS$4$0002 = true;
33    this.<wps>5__1 = this.<>4__this.GetNextWaypoint();
34    this.<wtw>5__2 = new WalkingToWaypoint(this.<wps>5__1);
35    while ((this.<wtw>5__2.ReachedWaypoint)
36    {
37        this.<>2__current = this.<wtw>5__2;
38        this.<>1__state = 2;
39        return true;
40    Label_00B4:
41        this.<>1__state = -1;
42        this.<tgt>5__3 = new Targetting();
43        this.<>2__current = this.<tgt>5__3;
44        this.<>1__state = 3;
45        return true;
46    Label_00E0:
47        this.<>1__state = 4;
48        if ((this.<tgt>5__3.FoundTarget && this.<>4__this.IsNode(this.<tgt>5__3.TargetName))
49        {
50            this.<wtn>5__4 = new WalkingToHarvestable(this.<tgt>5__3.TargetName, this.<tgt>5__3.Target.Location);
51            while ((this.<wtn>5__4.ReachedNode)
52            {
53                this.<>2__current = this.<wtn>5__4;
54                this.<>1__state = 4;
55                return true;
56            Label_0159:
57                this.<>1__state = -1;
58            }
59            this.<h>5__5 = new Harvesting();
60            while ((this.<h>5__5.DoneHarvesting)
61            {
62                this.<>2__current = this.<h>5__5;
63                this.<>1__state = 5;
64                return true;
65            Label_0198:
66                this.<>1__state = -1;
67            }
68        }
69        goto Label_01CB;
70    }
71 }
```

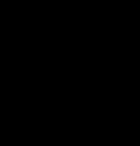
Reflected MoveNext() method

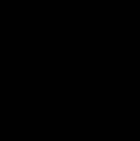
POSTED BY ERIK AT 5:47 PM
LABELS: C#, EVERHARVEST, WORKFLOW

6 COMMENTS:

 Chacko said...
Awesome post thanks a lot
JANUARY 29, 2010 AT 2:21 AM

Anonymous said...
Great explanation!
APRIL 1, 2010 AT 11:32 PM

 Chris Miller said...
That was a very nice explanation. I never really knew what yield was used for or what was under the hood.
APRIL 19, 2010 AT 12:51 PM

 Erik said...
Hey Chris,

Thanks => Bear in mind that my use of the Yield keyword in this post is somewhat unorthodox, although I find it's quite useful.
APRIL 19, 2010 AT 1:20 PM

Anonymous said...
I have to give you props on yet another use for the yield stmt. If I had considered that in times past, I wouldn't have had to build my own state machine for handling some game logic in XNA...kudos
JANUARY 26, 2011 AT 9:01 PM

 Gabriel Green said...
Nice post. I like 'alternate bool' use of yield. What's everharvest?
MARCH 2, 2011 AT 2:06 PM

Post a Comment


Newer Post


Home

Older Post

Subscribe to: Post Comments (Atom)

SUBSCRIBE

 Posts

 Comments

TAGS

c#
f#
eq2
everharvest
games
hunter bot
workflow
configuration
mvc
name generator
group bot
jquery
parsing
rant
ajax
asm
bdd
brainfck
evercraft
genetic algorithm
ling to sql
validation

BLOG ARCHIVE

► 2010 (6)

▼ 2009 (33)

► December (3)

► November (1)

► September (4)

► August (5)

► July (3)

► June (5)

► May (4)

► March (3)

▼ January (5)

I keep learning new things about C#

Progress

Little accomplishments

Yield, and the C# state machine

New Year, New Autocrat

► 2008 (18)