for Syntax, Linked Lists, Inheritance, Trees, IEnumerable and *foreach*

Introduction

This essay is definitely a mixed bag. The *for* statement in C/C++/C#/Java can do the things that a DO-loop in FORTRAN does, but is noticeably more general. So I thought I'd write a short document about the *for* statement. But to explain its power over simple DO-loops, I thought I'd show how it can be used with linked lists. But linked lists are a simple but useful example of OOP Inheritance. And *for* statements can also be used to walk a tree structure. But that generalizes into something called *IEnumerable*, which leads to the *foreach* keyword. Hence the mixed bag. Here goes…

for Statements – Basic Looping

I won't cover every possible variant of the *for* statement, but I'll talk about its main use of running code multiple times, incrementing (or decrementing) a counter each time. Then I'll show how it's more powerful than that.

Suppose you want to call subroutine *foo* 10 times, with a parameter ranging from 0..9. You could write

```
        int     i;
        for (i = 0; i < 10; ++i) {          // "++i" is the same as i = i + 1
                foo(i);
        }
```

The general syntax is

```
        for (loop init code; loop termination condition; on to next iteration) { … }
```

So the above is, in effect, expanded into, and exactly equivalent to

```
        int i;
        // Initialization code
        i = 0;
TopOfLoop:
        if (i >= 10) {              // Check for loop termination
                go to EndLoop:
        }
        // Body of Loop
        foo(i);
        // On to next iteration
        ++i;
        go to TopOfLoop;
EndLoop:
```

Side note: You don't need a separate declaration for *i*. It's normal to write

```
        for (int i = 0; i < 10; ++i) { … }
```

But if you do it this way, *i* goes out of lexical scope after the block is executed.

A few other examples…

```
for (int i = 1; i <= 10; ++i) { … }            // Not 0 to 9, but 1 to 10
for (int i = 0; i < 10; i += 2) { … }          // Even numbers from 0 to 8
for (int n = 10; n > 0; --n) { … }             // 10 downto 1
for (int x = 20; x < 10; ++x) { … }            // Empty loop
```

Note: A *for* loop doesn't strictly need braces following it. It takes a single statement after it, which can be a block of code encased in braces. It's quite valid to write

```
for (int n = 0; n < 10; ++n)
       foo(n);
```

But this runs into trouble if you write

```
for (int n = 0; n < 10; ++n)
       foo(n);
       MessageBox.Show("Just called foo");
```

The MessageBox will show up only once, not 10 times. It's basically a source code formatting issue, but it can be an absolute pain to track down. Without the braces, the body of the *for* will be only the next statement, *foo(n);* I always use braces, even if the loop body consists of only a single statement. Belt and suspenders[1] and all that.

Leaving the loop early

The *break* and *continue* statements help here.

break means to immediately exit the loop.

continue doesn't execute the rest of the body of the loop, but immediately goes onto the next iteration.

The following code will call foo(0), foo(1), foo(3) and foo(4), skipping calling foo(2).

```
for (int n = 0; n < 10; ++n) {
       if (n >= 5) break;
       if (n == 2) {
              continue;
       }
       foo(n);
}
```

++n and n++

Side note: *++n* increments n and returns the updated value. It's equivalent to *n = n + 1*, which in turn is more often written as *n += 1*.

[1] Quite suitable. A synonym for suspenders is *braces*!

n++ also increments n. The difference is that it returns the *original* value of *n*. So using *print* as pseudo-code,

```
int n = 5;
print n;                    // 5 - duh
print ++n;                  // 6
print n++;                  // Also 6 (!)
print n;                    // 7
```

When *n++* is used as the update clause in a *for* loop, the return value isn't used (as seen in the expanded version of a *for* statement above). So the following two statements are compiled identically. Which one you use is a matter of taste.

```
for (int n = 0; n < 10; ++n)
for (int n = 0; n < 10; n++)
```

for Statements – Looping Through a Linked List

Suppose you have a simple LinkedList class that (ignoring ctors, etc) looks like…

```
class LinkedList {
        public int          n;            // Each node has a simple int
        public LinkedList   NextNode;     // Reference to next node in the list
                                          // or null meaning the end of the list
}
```

Also assume that you've got a standard anchor that points to the first element in the list, or null if the list is empty. To traverse the list, we'd normally write

```
LinkedList    Anchor;                     // Set somewhere
…
for (LinkedList ptr = Anchor; ptr != null; ptr = ptr.NextNode) {
        foo(ptr.n);
}
```

So here's the power of *for*. It's generalized the concept of a loop from a mere counter to explicit *Initialize, Test For End, On to Next* expressions.

Linked Lists via Inheritance

While there is a LinkedList<> class[2] (in System.Collections.Generic), let's ignore that for now and write our own[3]. Suppose you want a linked list of, say, class Book. While you *could* define it as above, replacing the *int n* with Book MyBook, a much better[4] way to organize things would be…

[2] The system-supplied class is actually a doubly-linked circular list and is more sophisticated than our simple example here.
[3] Which we could if we put it into our namespace. Or didn't include *using System.Collections.Generic*.
[4] Better in the sense that we have the opportunity to *re-use* our LinkedList class, unchanged, in another project.

```
class LinkedList<T> {
	public LinkedList<T>	NextNode;
	// Suitable constructor etc here
}
class MyLinkedList : LinkedList<TData> {
	public TData data;  // Or whatever instance name you'd like
}
```

So the *LinkedList* class (I'll get around to the <T> syntax in a minute) has exactly one field in it, a link to the next node in the list.

But when *MyLinkedList* inherits from *LinkedList*, it now has two fields in it, the data and the link.

So we've abstracted away the concept of a linked list, put it into its own class, and can now derive whatever secondary classes we want from it.

LinkedList<T>

This is a C# language feature called *Generics*. In (very!) brief, it's a macro-like feature. The *T*[5] represents a data type when the class is instantiated. Any reference to *T* is replaced by whatever data type the user specifies.

So, for example, the (non-linked)List<> class is a vector that automatically grows when you append a new element. List<int> represents a vector of int's. List<string> is a list of strings. List<Book> is a list of instances of class Book. There are at least two advantages to this.

- It's type safe. If you try to add a non-*int* to a List<int> (or non-*Book* to a List<Book>, etc), then the compiler will complain[6], thus guaranteeing that at runtime there's nothing in our list but *int*s (*Book*s, etc).
- There's the opportunity for optimization. For example, the compiler can generate efficient code if it knows that this is a vector of *int*s (*float*s, etc).

We create, say, a list of *int*s then referencing them using the syntax

```
var nums = new List<int>();       // New, empty vector
nums.Add(5);                      // Append 5 to the end of the vector
nums.Add(12);
int sum = nums[0] + nums[1];
```

Trees

Getting back to the *for* statement, suppose you have a tree data structure and you want to visit each node in turn. There are three standard ways to do so, in-order, pre-order and post-order. See

[5] It doesn't have to be called *T*, it can be anything you like. *T* is often used because it's the first letter of *Type*.
[6] But remember that a *Manager* is-a *Employee*, so a List<*Employee*> can have a *Manager* object added to it.

http://en.wikipedia.org/wiki/Tree_traversal for details. Ignoring for now exactly which order you'd like to traverse the tree in, wouldn't it be nice if you could write

```
for (TreeNode ptr = TreeRoot; ptr != null; ptr = ptr.NextNode) { … }
```

Look familiar?

IEnumerable

.NET has generalized the concept of going from one item in a collection (array, List<>, LinkedList<>, Tree, etc, etc, etc) via a set of methods called an *Enumerator*.

An Enumerator has the following three methods / properties:

- *Current* – A property that returns the current item in the collection you're looping through.
- *MoveNext*() – A method that knows how, given the Current item, to get to the next item. It returns true or false, depending on whether Current was the last element of the collection.
- *Reset*() – Starts (or starts over), setting Current to the first element in the collection.

So for our *for* loop above to work on a tree, it would need to implement those two methods and the *Current* property. It's up to the author of the class to figure out what's required to implement these. But these details are usually of no interest to the user (programmer). He just wants to write the *for* loop above.

C# has the concept of an i*nterface*. This is a bit like a class definition except that

- It has no instance variables
- It has method / property declarations, but without specifying any implementation code

Its syntax for the above would be

```
public interface IEnumerator[7]<T> {
        T Current { get; }
        bool MoveNext();
        void Reset();
}
```

If you derive a class from an interface, you're guaranteeing that this class will have these methods / properties.

While a class can only inherit from a single base class[8], it can "inherit" from any number of interfaces.

So our definition of the LinkedList<T> class should really be

[7] By convention, the names of interfaces start with "I".
[8] But the base class could inherit from a single super-base class, which in turn could inherit from a single super-super-class, etc. But any given class can have only one direct ancestor.

```
class LinkedList<T> : IEnumerable<T> {
      public LinkedList<T>      NextNode;
      // Suitable constructor etc here
}
```

And the author of this class would have to supply a *Current* property and *Reset*() and *MoveNext*() methods.

foreach

The *foreach* statement operates on anything that supports IEnumerable<T>. For example, System.Diagnostics.Process.GetProcesses() returns a simple array of Process instances. Arrays implement IEnumerable[9].

```
foreach (Process proc in System.Diagnostics.Process.GetProcesses()) {
      Console.WriteLine("Process {0} has a working set of {1}",
            proc.Name, proc.WorkingSetSize);
}
```

And our LinkedList example could be

```
foreach (var item in MyLinkedList) { … }
```

And our Tree example above could be something like

```
foreach (TreeNode node in MyTree.Nodes) {
      Visit(node);
}
```

Finally, notice that *foreach* may allow you to change your data structures with minimal recoding. For example, switching from a fixed-sized array, to a List<>, to a LinkedList<> and maybe even to a Tree<> wouldn't change your *foreach* at all. Yeah, they use different algorithms to go from one item to the next, but those details are hidden from you.

[9] A .NET array isn't a mere vector of values. It has additional information associated with it, such as the number of elements in the array. This is so, for example, the runtime can check for subscripts out of bounds. So a simple array is implemented as a class, and thus can have methods associated with it.