LINQ – Language-Integrated Query Introduction

Contents

Introduction

LINQ is a feature of the C#[1] language introduced in C# 3.0. It provides a SQL-like statement facility for a variety of data types. The final syntax depends on a number of other language features, each of which has its own uses. But the set of these features, added together, leads to a very useful language feature.

This document will describe the individual pieces, then at the end will tie them all together and show how they lead to LINQ.

Note: After you read this, I highly recommend you look at *The Evolution Of LINQ And Its Impact On The Design Of C#* at http://msdn.microsoft.com/en-us/magazine/cc163400.aspx

[1] It was also added to VB.NET at the same time.

The *var* Keyword

Suppose you were to write "int n = 6;". When you think about it, the *int* keyword is a bit redundant. *6* is an *int*, so you're setting *n* to an *int* and you shouldn't have to declare it as an *int*. And with the *var* keyword, you don't have to. *var* infers the data type of *n* from the initialization expression. So "var n = 6;" is *exactly* the same as "int n = 6;".

Now for *int*s, this isn't a big deal. In fact it's probably overkill. Just write "int n = 6;". But suppose you want to have a Dictionary (a type of hash table) that takes, say a key of type *string* (maybe an ISBN) and returns a *Book*. We could write, either way…

```
Dictionary<string, Book> BookFromIsbn = new Dictionary<string, Book>();
var BookFromIsbn = new Dictionary<string, Book>();
```

These two statements would compile exactly the same.

Please note that, unlike some other languages (e.g. JavaScript), the *var* keyword does *not* imply that the data type is dynamically determined at runtime. It is still 100% statically typed at compile time. If you use *var*, you *must* have an initialization expression from which to infer the data type.

IEnumerable<T>

As we've discussed in another document (*for Syntax, Linked List, Inheritance, Trees, IEnumerable and foreach.docx*), if a class inherits from IEnumerable<T>, it promises to a have a property called *Current*, and methods *MoveNext* and *Reset* that together allow a set of things to be walked through. Assuming the class author provides these for her class, you can enumerate very disparate things. This could include (but is by no means limited to):

- Arrays
- List<T>
- LinkedList<T>
- A Tree structure
- Items in a database
- Each line in a text file
- The list of processes running on the current machine
- The list of all machines on your network[2]
- The list of all hardware devices on a machine on your network (potentially down to the level of the manufacturing dates of the RAM chips on the motherboard![3]).
- etc, etc, etc.

[2] Of course you need adequate permissions and software tools for some of these things.

[3] For years Windows has supported the Windows Management Instrumentation (WMI) protocol. SANDRA (a highly recommended utility at http://www.sisoftware.net/) undoubtedly uses it. And .NET has the System.Management namespace to talk to WMI. See http://msdn.microsoft.com/en-us/library/ms257340(vs.80).aspx

Anonymous Classes

Although it might sound obscure at first, as we'll see later, we'll want to define an instance of a class but without giving the class a name. But then how do we refer to the instance. *var* to the rescue[4]. We can write…

```
var MyAnonymousClassInstance = new { FirstName = "Jane", LastName = "Doe" };
```

We can now refer to MyAnonymousClassInstance.LastName.

The compiler will assign an otherwise invalid name to the anonymous class (e.g. perhaps "$1`0001" or some such). So the class will indeed have a name, just one that you couldn't validly use in your program. Which is why you need the *var* keyword for this.

Note: The compiler will infer the data types from the initialization expressions.

Extension Methods – Part I

Let's take the String[5] class as an example. It offers many methods for manipulating strings. You can convert a string to upper- or lower-case. You can pad the string on the left or right to a certain width. Of course you can do substrings. And there are many other methods to manipulate strings.

But suppose you want something custom to your application. Suppose you want to capitalize the first letter of each word in a string. But there are issues to consider. Maybe some of your data is all in lower case, some all in upper case, and even some with mixed case in the middle of a word! Oh, and let's coalesce multiple embedded blanks inside the string into a single blank. Let's call this Canonicalizing the string and assume you've written a method that takes a string and returns a Canonicalized string[6]. How would we use it?

```
string s = "   NOW IS    the time For aLL  GoOd Men to come.  ";
string s2 = Canonicalize(s); // s2 = "Now Is The Time For All Good Men To Come."
```

OK, that works. But it has a couple of drawbacks. The one I'll talk about here has to do with ease of use. All our other string functions tend to be *fluent*, which means they take a string and return a string, letting us string[7] multiple string methods together (if we need to). For example, the following works…

```
string s2 = s.Trim().ToUpper().Substring(2, 6).PadRight(10);
```

But we *couldn't* write

```
string s2 = s.Trim().ToUpper().Substring(2, 6).PadRight(10).Canonicalize();
```

[4] This is our first hint as to how these seemingly disparate language additions will add up to something greater than the sum of their parts.
[5] The *string* data type in C# is a synonym for the *String* (note capitalization) class in the Framework.
[6] Homework: Write such a method. For bonus marks, have a list of "noise" words (e.g. a, an, the, in, of, etc) that should not be capitalized.
[7] Pun definitely intended.

And for that matter we couldn't even write

```
string s2 = s.Canonicalize();
```

The *Canonicalize* method isn't a member of the *String* class; it's a member of whatever class you've written.

Could we take a copy of the *String* class and modify it to include our *Canonicalize* method? Well, technically yes, but you couldn't (for copyright reasons) distribute your modified *String* class to others. And when a new version of the Framework came out, with a new version of the *String* class, you'd have to modify it each time. And so on. So that's out.

Ah, but maybe we can create a new *MyString* class, inheriting from *String* and including *Canonicalize*. Well, if we could, then any strings we'd want to use *Canonicalize* with would have to be declared as *MyString foo;* not just *string foo;* and this would clearly lead to confusion with perhaps some string variables declared as *string*, and others as *MyString*.

But most importantly, the definition for *String* is marked as *sealed*. This is a C# keyword that says you can't derive from the specified class. Perhaps a bit of a nuisance, but the rationale as to why many of the Framework classes are sealed can be found in the Eric Lippert[8] article at http://blogs.msdn.com/b/ericlippert/archive/2004/01/22/61803.aspx.

The good news is that with Extension Methods, there's now a way that pretty well gives you what you want. But first we have to talk about the *this* parameter to methods.

The *this* Parameter

Suppose I write

```
var Pat = new Person("Pat");
var Wes = new Person("Wes");
```

and the *Person* class has a *public string Name* field and a simple *PrintName* method. I could write

```
Pat.PrintName();
Wes.PrintName();
```

This would work, but how does the *PrintName* method (which (with its *Person* class, of course) could be off in a DLL somewhere) work when it has never heard of your *Pat* or *Wes* variable names (and thus the two *Person* instances)?

The answer is that every (with one exception, described shortly) method in a class has, in addition to whatever parameters you've defined for the method, an additional invisible parameter at the beginning called *this*. It is a reference to the instance variable (in this case *Pat* or *Wes*, of data type *Person*) we're

[8] An ex-Waterloonie and former member of the Microsoft C# compiler team.

calling the method on. So within *PrintName*, any reference to field *Name* is implicitly qualified to be *this.Name*.

It looks like the *PrintName* method has 0 parameters. But technically it has 1, the *this* parameter. It's as if it were defined as

```
public void PrintName(Person this);
```

The exception to this invisible parameter rule is if the method is marked as *static*. In that case there is no *this* reference passed. When would you want something like this? Consider the square root method. It's a *static* method of the *static Math* class and you'd write

```
var Sqrt_2 = Math.Sqrt(2.0);
```

Extension Methods – Part II

To make our *Canonicalize* method such that it can be called directly on a *string*, we'll use a special syntax.

```
public static class MyStringExtensions {
	public static string Canonicalize(this string s) {
		// The real routine would be more complex than this.
		// For now, just return the string in upper case.
		return s.ToUpper();
	}
}
```

Note the *this* keyword in the parameter list. Also note that that both the method and the class it's defined in are marked as *static*.

When the compiler sees a method in a static method in a static class that has its first parameter qualified by the *this* keyword, it's considered to be an extension to the specified parameter type (in this case *string*). So now you can extend classes with your own methods, without inheritance and you would be able to write:

```
string s2 = s.Canonicalize();
```

The compiler would internally rewrite this to be

```
string s2 = MyStringExtenstion.Canonicalize(s);
```

Extension methods are particularly useful with Generics. For example,

```
public static class MyExtensions {
	public static string Visit(this string s) {
		// The real routine would be more complex than this. For now, just return
```

```
            // the string in upper case.
            return s.ToUpper();
        }
}
```

TODO: Talk about extension methods, such as .Average, .First, .Max, etc, etc, etc.

TODO:

Strings are Immutable

Just a side note: An instance of a *string* cannot be changed. In other languages (e.g. C), since a string can be considered to be an array of *char*acters, we could write

```
s[3] = 'X';                 // Gives compile-time error in C#
```

to change the 3rd character of *s* to an X. But C# doesn't allow strings to be modified. There are several reasons but one has to do with immutability. See http://en.wikipedia.org/wiki/Immutable_object

So be careful. A common beginner's mistake is to write something like

s.ToUpper();

and expect *s* to be changed to all upper case. *The* ToUpper *method doesn't change its object; it returns a new* string.

If you want *s* to be changed, write

```
s = s.ToUpper();
```

and similarly for other methods (e.g. Replace).

Delegates – Named

A *delegate* is essentially a type-safe function pointer object. Named delegates were introduced in C# 1.0 and anonymous delegates came in 2.0. But as of C# 3.0, *lambda expressions* (see below) have mostly supplanted delegates[9].

Delegates are still fully supported to allow older programs that used them to still compile and run. But new programs should almost always use lambda expressions.

I did write sections on both Named and Anonymous delegates. But since these have mostly been supplanted by lambdas, the main use of explicit delegates[10] is in interfacing with legacy code (including

[9] In fact, Microsoft is on record as saying that if they'd known that lambda expressions were coming, they never would have introduced anonymous delegates.

methods introduced in the .NET Framework before lambdas were introduced). So I've kept the text in, colored in red, but you can pretty well ignore all of it.

```
public delegate void WriteDel(string message);
```

would define a data type called *WriteDel* that contained a pointer to a function that took a *string* as a parameter and returned *void*. Suppose you wanted to have debug information written either to stdout or to a database table, depending on a flag. One way to approach this would be as follows…

```
void WriteToStdOut(string message) {
       Console.WriteLine(message);
}
void WriteToDatabase(string message) {
       // Add code here to write to the database
}
…
WriteDel Output;
if (bOutputToConsole == true) {
       Output = WriteToStdOut;
} else {
       Output = WriteToDatabase;
}
…
Output("I wonder where this message will wind up!");
```

Delegates – Anonymous

Sometimes you don't want to bother defining a prototype (*WriteDel* above) and giving it a name. Sometimes you would just like to say, "Here's the code I want to use".

Consider the *qsort* function in the C runtime library. It takes a pointer to an array, the number of items in the array and the size of each. But it has one more parameter and that's a pointer to a comparison function. Whenever the *qsort* routine needs to compare two items, it calls (through the pointer) the user-supplied comparison routine that returns whether the first parameter (to the comparison routine) is less than, equal to, or greater than the second.

Do you *need* this comparison function? Well, in general, yes. Even if you're sorting a simple vector of *int*s, are you sorting them ascending or descending? You need two comparison functions to support both modes.

More generally, suppose you're trying to sort, not simple *int*s, but a vector of *struct*s of type *Person* that have, say, a Name and an Age field. And if you sometimes need to sort by Name and sometimes by Age, then you need two[11] comparison routines you can pass to *qsort*.

In .NET, we don't need explicit counts and lengths of a collection (e.g. array) class. We could write

[10] delegates are used as the basis for the Func and Action types (see the section on Lambda Expressions below), but these days new code will seldom use the delegate keyword explicitly in its code.
[11] Four, if you need both ascending and descending sorts on both fields.

```
var MyList = new List<int> { 5, 3, 12, 6 };
MyList.Sort();                                  // Defaults to ascending sort

MyList.Sort(                                    // Sort MyList ascending
       delegate(int x, int y) {
              if (x < y) return -1;
              if (x == y) return 0;
              return 1;
       }
);
MyList.Sort(                                    // Sorts MyList descending
       delegate(int x, int y) {
              if (y < x) return -1;
              if (y == x) return 0;
              return 1;
       }
);
```

So basically, an anonymous delegate has a parameter list and a code body (and a return type, which is inferred), but no name. And that's because in such cases, we never want to refer to this body of code again, so why bother going to the trouble of giving it one?

With all this said, you can forget about anonymous delegates. They were defined in C# 2.0, but C# 3.0 introduced *lambda expressions*. Microsoft is on record as saying that if they knew they were going to add lambda expressions to the language, they would never have defined anonymous delegates. But once a feature is added to a language, you can't break existing code, so they're still supported.

So what are lambda expressions? I'm glad you asked!

Lambda (=>) Expressions

In the 1930's, mathematical logician Alonzo Church invented something called the Lambda Calculus, using the Greek letter *lambda* (λ) to introduce a calculation and its constituent parameters. It was later used by John McCarthy in the late 1950's in his LISP language.

Consider the *qsort* function in the C runtime library. It takes a pointer to an array, the number of items in the array and the size of each. But it has one more parameter and that's a pointer to a comparison function. Whenever the qsort routine needs to compare two items, it calls (through the pointer) the user-supplied comparison routine that returns whether the first parameter (to the comparison routine) is less than, equal to, or greater than the second.

Do you need this comparison function? Well, in general, yes. Even if you're sorting a simple vector of ints, are you sorting them ascending or descending? You need two comparison functions to support both modes.

More generally, suppose you're trying to sort, not simple *int*s, but a vector of *struct*s of type *Person* that have, say, a Name and an Age field. And if you sometimes need to sort by Name and sometimes by Age, then you need two[12] comparison routines you can pass to qsort.

The syntax of a lambda expression is basically *paramter-list => code*.

In .NET, we don't need explicit counts and lengths of a collection (e.g. array) class. We could write

```
var MyList = new List<int> { 5, 3, 12, 6 };
MyList.Sort();                                  // Defaults to ascending sort
MyList.Sort(                                    // Sort MyList ascending
        (x, y) => {                             // Parameter list to left of =>
              if (x < y) return -1;
              if (x == y) return 0;
              return 1;
       }
);

MyList.Sort(                                    // Sorts MyList descending
        (x, y) => {
              if (y < x) return -1;
              if (y == x) return 0;
              return 1;
       }
);
```

Perhaps the first thing we notice is that we don't declare the types of the parameters *x* and *y*. The compiler is smart enough to realize (through Generics behind the scenes) that you're working on a List<int>, so *x* and *y* must be *int*s.

The *Func* data type represents a lambda expression that returns a value. The *Action* data type is used for lambdas that don't return a value (*void* return).

There's a bit more to lambda syntax than the above, but this will do for now.

yield – How to Write a Method That Returns an IEnumerable<T>

In the first version of the .NET Framework, to have a method return enumerable data, you had to go to a fair amount of work, defining an instance of an Enumerator class, with Reset() and MoveNext() methods and a Current property.

This was made much easier in C# 2.0 with the introduction of the *yield* keyword. The compiler would take generate (behind the scenes) the code necessary to make it work[13].

```
IEnumerable<int> GetVals() {
       int[] vals = { 5, 7, 12, 3 };
```

[12] Four, if you need both ascending and descending sorts on both fields.
[13] It would even generate a hidden class and a small Finite State Machine to keep track of where you were in your processing!

```
        foreach (int val in vals) {
                yield return val;
        }
}
```

And call it via

```
foreach (var item in GetVals()) {
        Console.WriteLine(item);
}
```

Also

```
IEnumerable<string> GetLines(string Filename) {
        var InFile = new StreamReader(Filename);
        string line;
        while ((line = InFile.ReadLine()) != null ) {
                yield return line;
        }
        InFile.Close();
}
```

And invoke that via

```
foreach (var line in GetLines(@"C:\LRS\TestFile.txt")) {
        Console.WriteLine(line);
}
```

There's more to the *yield* keyword, but this should give you the idea.

Examples of Extension Methods With Lambdas

TODO: Start with WHERE

Object Initializers

Object initializers let you assign values to any accessible fields or properties of an object at creation time without having to invoke a constructor followed by lines of assignment statements. For example,

```
Cat cat = new Cat { Age = 10, Name = "Fluffy" };
```

You don't even need an existing class. You could write

```
var pet = new { Age = 10, Name = "Fluffy" };
```

and the compiler would define an anonymous class and assign values to the field names you supply.

Is this useful? Out of context, probably not too much. But as we'll see later, it will come in handy.

LINQ (Language Integrated Query)

LINQ Syntax

Remember, LINQ stands for Language Integrated Query. That's *Query* as in *Structured Query Language (SQL)*, where a query is a request for data from a data source, optionally filtered, sorted, grouped, summarized and so forth.

Let's start off by taking a look at what a LINQ statement would look like.

```
var procs = System.Diagnostics.Process.GetProcesses();
var BigCpuUsers = from p in procs
                  where p.TotalProcessorTime > new TimeSpan(0, 0, 10)     // 10 seconds
                  orderby p.TotalProcessorTime descending
                  select new { p.ProcessName, p.MainModule.FileName,
                          p.UserProcessorTime,
                          p.PrivilegedProcessorTime, p.WorkingSet };

foreach (var proc in BigCpuUsers) {
       Console.WriteLine("Name = {0}, WS = {1}, User = {3}, Priv = {2}, Total = {4},
              Filename = {5}",
              proc.ProcessName, proc.WorkingSet, proc.PrivilegedProcessorTime,
              proc.UserProcessorTime,
              proc.UserProcessorTime + proc.PrivilegedProcessorTime, proc.FileName);
}
```

Why FROM first? TODO:

LINQ to Almost Anything

TODO: