Edulinq: Reimplementing LINQ to Objects

This constitutes the text of the bodies of the blog posts; please see the original posts for community comments etc.

Note that the posts have *not* been particularly edited for out-of-blog reading: some will no doubt refer to comments, or timescales, or generally indicate their origin. Consider this a somewhat quick-and-dirty hack to make the posts available in other forms.

Part 1 - Introduction

About a year and a half ago, I gave a talk at a DDD day in Reading, attempting to reimplement as much of LINQ to Objects as possible in an hour. Based on the feedback from the session, I went far too fast... and I was still a long way from finishing. However, I still think it's an interesting exercise, so I thought I'd do it again in a more leisurely way, blogging as I go. Everything will be under the "Eduling" tag, so that's the best way to get all the parts in order, without any of my other posts.

General approach

The plan is to reimplement the whole of LINQ to Objects, explaining each method (or group of methods) in a blog post. I'm going to try to make the code itself production quality, but I'm not going to include any XML documentation - if I'm already writing up how things work, I don't want to do everything twice. I'll include optimizations where appropriate, hopefully doing better than LINQ to Objects itself.

The approach is going to be fairly simple: for each LINQ method, I'll write some unit tests (most of which I won't show in the blog posts) and make sure they run against the normal .NET implementation. I'll then comment out the using directive for System.Linq, and introduce one for JonSkeet.Linq instead. The tests will fail, I'll implement the methods, and gradually they'll go green again. It's not quite the normal TDD pattern, but it works pretty well.

I will write up a blog entry for each LINQ operator, probably including all of the production code but only "interesting" tests. I'll highlight important patterns as they come up - that's half the point of the exercise, of course.

At the end of each post, I'll include a link to download the "code so far". For the sake of anyone looking at these posts in the future, I'm going to keep the downloads numbered separately, rather than updating a single download over time. I'm hoping that the code will simply grow, but I dare say there'll be some modifications along the way too.

The aim is *not* to end up with LINQBridge: I'm going to be targeting .NET 3.5 (mostly so I can use extension methods without creating my own attribute) and I'm certainly not going to start worrying about installers and the like. The purpose of all of this is purely educational: if you follow along with these blog posts, with any luck you'll have a deeper understanding of LINQ in general and LINQ to Objects in particular. For example, topics like deferred execution are often misunderstood: looking at the implementation can clarify things pretty well.

Testing

The unit tests will be written using NUnit (just for the sake of my own familiarity). Fairly obviously, one of the things we'll need to test quite a lot is whether two sequences are equal. We'll do this using the TestExtensions class from MoreLINQ (which I've just copied into the project). The netbook on which I'll probably write most of this code only has C# Express 2010 on it, so I'm going to use the external NUnit GUI. I've set this in the project file as the program to start... which you can't do from C# Express directly, but editing the project file is easy, to include this:

```
<StartAction>Program</StartAction>
<StartProgram>C:\Program Files\NUnit-2.5.7.10213\bin\net-2.0\nunit-
x86.exe</StartProgram>
```

It's a bit of a grotty hack, but it works. The "additional command line parameters" are then set to just JonSkeet.Linq.Tests.dll - the current directory is the bin/debug directory by default, so everything's fine. Obviously if you want to run the tests yourself and you have ReSharper or something similar, you can see the results integrated into Visual Studio.

Although I'm hoping to write reasonably production-level code, I doubt that there'll be as many unit tests as I'd *really* write against production code. I fully expect the number of lines of test code to dwarf the number of lines of production code even so. There are simply vast numbers of potential corner cases... and quite a few overloads in several cases. Remember that the goal here is to examine interesting facets of LINQ.

Code layout

Just like the real LINQ to Objects, I'll be creating an enormous static Enumerable class... but I'll do so using partial classes, with one method name (but multiple overloads) per file. So Where will be implemented in Where.cs and tested in WhereTest.cs, for example.

First code drop

The first zip file is available: Linq-To-Objects-1.zip. It doesn't contain any production code yet - just 4 tests for Where, so I could check that NUnit was working properly for me. Next stop... implementing Where.

Back to the table of contents.

Part 2 - Where

Warning: this post is quite long. Although I've chosen a simple operator to implement, we'll encounter a few of the corner cases and principles involved in LINQ along the way. This will also be a somewhat experimental post in terms of format, as I try to work out the best way of presenting the material.

We're going to implement the "Where" clause/method/operator. It's reasonably simple to understand in general, but goes into all of the deferred execution and streaming bits which can cause problems. It's generic, but only uses one type parameter (which is a big deal, IMO - the more type parameters a method has, the harder I find it to understand in general). Oh, and it's a starting point for query expressions, which is a bonus.

What is it?

"Where" has two overloads:

```
public static IEnumerable<TSource> Where(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)

public static IEnumerable<TSource> Where(
    this IEnumerable<TSource> source,
    Func<TSource, int, bool> predicate)
```

Before I go into what it actually does, I'll point out a few things which are going to be common across nearly all of the LINQ operators we'll be implementing:

- These are *extension methods* - they're declared in a top-level, non-nested, static class and the first parameter has a "this" modifier. They can be invoked roughly as if they were instance methods on the type of the first parameter.
- They're *generic methods* - in this case there's just a single type parameter (TSource) which indicates the type of sequence we're dealing with. So for (say) a list of strings, TSource would be string.
- They take *generic delegates* in the Func<...> family. These are usually specified with lambda expressions, but any other way of providing a delegate will work fine too.
- They deal with *sequences*. These are represented by IEnumerable<T>, with an iterator over a sequence being represented by IEnumerator<T>.

I fully expect that most readers will be comfortable with all of these concepts, so I won't go into them in more detail. If any of the above makes you nervous, please familiarise yourself with them before continuing, otherwise you're likely to have a hard time.

The purpose of "Where" is to filter a sequence. It takes an input sequence and a *predicate*, and returns another sequence. The output sequence is of the same element type (so if you put in a sequence of strings, you'll get a sequence of strings out) and it will only contain elements from the input sequence which pass the predicate. (Each item will be passed to the predicate in turn. If the predicate returns true, the item will be part of the output sequence; otherwise it won't.)

Now, a few important details about the behaviour:

- The input sequence is *not* modified in any way: this isn't like List<T>.RemoveAll, for example.
- The method uses *deferred execution* - until you start trying to fetch items from the output sequence, it won't start fetching items from the input sequence
- Despite deferred execution, it will validate that the parameters aren't null immediately
- It *streams* its results: it only ever needs to look at one result at a time, and will yield it without keeping a reference to it. This means you can apply it to an infinitely long sequence (for example a sequence of random numbers)
- It will iterate over the input sequence exactly once each time you iterate over the output sequence
- Disposing of an iterator over the output sequence will dispose of the corresponding iterator over the input sequence. (In case you didn't realise, the foreach statement in C# uses a try/finally block to make sure the iterator is always disposed however the loop finishes.)

Many of these points will be true for a lot of our other operators too.

The overload which takes a Func<TSource, int, bool> lets the predicate use the index within the sequence as well as the value. The index always starts at 0, and increments by 1 each time regardless of previous results from the predicate.

What are we going to test?

Ideally, we'd like to test *all* of the points above. The details of streaming and how many times the sequence is iterated over are frankly a pain to deal with, unfortunately. Given how much there is to implement already, we'll come back to those.

Let's have a look at some tests. First, here's a simple "positive" test - we're starting with an array of integers, and using a lambda expression to only include elements less than 4 in the output. (The word "filter" is omnipresent but unfortunate. It's easier to talk about "filtering out" elements than "including" them, but the predicate is expressed in a *positive* way.)

```
[Test]
public void SimpleFiltering()
{
    int[] source = { 1, 3, 4, 2, 8, 1 };
    var result = source.Where(x => x < 4);
    result.AssertSequenceEqual(1, 3, 2, 1);
}
```

I've kept the TestExtensions from MoreLINQ, despite NUnit coming with CollectionAssert. I find the extension methods easier to work with for three reasons:

- They're extension methods, which helps to reduce the clutter
- They can use a parameter array for the expected output, which makes the test simpler to express
- The message is clearer when the assertion fails

Basically, AssertSequenceEqual does what you'd expect it to - it checks that the actual result (usually expressed as the variable you call the method on) matches the expected result (usually expressed as a parameter array).

So far, so good. Now let's check argument validation:

```
[Test]
public void NullSourceThrowsNullArgumentException()
{
    IEnumerable<int> source = null;
    Assert.Throws<ArgumentNullException>(() => source.Where(x => x > 5));
}

[Test]
public void NullPredicateThrowsNullArgumentException()
{
    int[] source = { 1, 3, 7, 9, 10 };
    Func<int, bool> predicate = null;
    Assert.Throws<ArgumentNullException>(() => source.Where(predicate));
}
```

I'm not bothering to check the name within the ArgumentNullException, but importantly I'm testing that the arguments are being validated immediately. I'm not trying to iterate over the result - so if the validation is deferred, the test will fail.

The final interesting test for the moment is also around deferred execution, using a helper class called ThrowingEnumerable. This is a sequence which blows up with an InvalidOperationException if you ever try to iterate over it. Essentially, we want to check two things:

- Just calling Where doesn't start iterating over the source sequence
- When we call GetEnumerator() to get an iterator and then MoveNext() on that iterator, we *should* start iterating, causing an exception to be thrown.

We'll need to do something similar for other operators, so I've written a small helper method in ThrowingEnumerable:

```
internal static void AssertDeferred<T>(
    Func<IEnumerable<int>, IEnumerable<T>> deferredFunction)
{
    ThrowingEnumerable source = new ThrowingEnumerable();
    var result = deferredFunction(source);
    using (var iterator = result.GetEnumerator())
    {
        Assert.Throws<InvalidOperationException>(() => iterator.MoveNext());
    }
}
```

Now we can use that to check that Where really defers execution:

```
[Test]
public void ExecutionIsDeferred()
{
    ThrowingEnumerable.AssertDeferred(src => src.Where(x => x > 0));
}
```

These tests have all dealt with the simpler predicate overload - where the predicate is only passed the item, not the index. The tests involving the index are very similar.

Let's implement it!

With all the tests passing when running against the real LINQ to Objects, it's time to implement it in our production code. We're going to use iterator blocks, which were introduced in C# 2 to make it easier to implement IEnumerable<T>. I have a couple of articles you can read if you want more background details... or read chapter 6 of C# in Depth (either edition). These give us deferred execution for free... but that can be a curse as well as a blessing, as we'll see in a minute.

At its heart, the implementation is going to look something like this:

```
// Naive implementation
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    foreach (TSource item in source)
    {
        if (predicate(item))
        {
            yield return item;
        }
    }
}
```

Simple, isn't it? Iterator blocks allow us to write the code pretty much how we'd describe it: we iterate over each item in the source, and if the predicate returns true for that particular item, we yield (include) it in the output sequence.

Lo and behold, some of our tests pass already. Now we just need argument validation. That's easy, right? Let's give it a go:

```
// Naive validation - broken!
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    if (predicate == null)
    {
        throw new ArgumentNullException("predicate");
    }
    foreach (TSource item in source)
    {
        if (predicate(item))
        {
            yield return item;
        }
    }
}
```

Hmm. Our validation tests still seem to be red, and putting a breakpoint on the "throw" statements doesn't help... they're not getting hit. What's going on?

I've given a few pretty broad hints already. The problem is deferred execution. Until we start trying to iterate over the result, *none of our code will run*. Our tests deliberately *don't* iterate over the result, so validation is never performed.

We've just hit a design flaw in C#. Iterator blocks in C# simply don't work nicely when you want to split execution between "immediate" (usually for validation) and "deferred". Instead, we have to split our implementation into two: a normal method for validation, which then calls the iterator method for the deferred processing:

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    if (predicate == null)
    {
        throw new ArgumentNullException("predicate");
    }
    return WhereImpl(source, predicate);
}

private static IEnumerable<TSource> WhereImpl<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    foreach (TSource item in source)
    {
        if (predicate(item))
        {
            yield return item;
        }
    }
}
```

It's ugly, but it works: all our index-less tests go green. From here, it's a short step to implement the version using an index as well:

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<T, int, bool> predicate)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    if (predicate == null)
```

```
    {
        throw new ArgumentNullException("predicate");
    }
    return WhereImpl(source, predicate);
}

private static IEnumerable<TSource> WhereImpl<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int, bool> predicate)
{
    int index = 0;
    foreach (TSource item in source)
    {
        if (predicate(item, index))
        {
            yield return item;
        }
        index++;
    }
}
```

Now the bar is green, and we're done. Hang on though... we haven't used it every way we can yet.

Query expressions

So far, we've been calling the method directly (although as an extension method) - but LINQ also provides us with query expressions. Here's our "SimpleFiltering" test rewritten to use a query expression:

```
[Test]
public void QueryExpressionSimpleFiltering()
{
    int[] source = { 1, 3, 4, 2, 8, 1 };
    var result = from x in source
                 where x < 4
                 select x;
    result.AssertSequenceEqual(1, 3, 2, 1);
}
```

(Note that the name is different here to in the downloadable code, to stop the blog server software blocking the name of the method. Grr.)

That will produce *exactly* the same code as our earlier test. The compiler basically translates this form into the previous one, leaving the condition (x < 4) as a lambda expression and then converting it appropriately (into a delegate in this case). You may be surprised that this works as we have no Select method yet... but in this case

we have a "no-op" select projection; we're not actually performing a real transformation. In that case - and so long as there's *something* else in the query, in this case our "where" clause - the compiler effectively omits the "select" clause, so it doesn't matter that we haven't implemented it. If you changed "select x" to "select x * 2", it would fail to compile against our Where-only LINQ implementation.

The fact that query expressions are just based on patterns like this is a very powerful feature for flexibility - it's how LINQ to Rx is able to only implement the operators that make sense in that environment, for example. Similarly, there's nothing in the C# compiler that "knows" about IEnumerable<T> when it comes to query expressions - which is how entirely separate interfaces such as IObservable<T> work just as well.

What have we learned?

There's been a lot to take in here, in terms of both implementation and core LINQ principles:

- LINQ to Objects is based on extension methods, delegates and IEnumerable<T>
- Operators use deferred execution where appropriate and stream their data where possible
- Operators don't mutate the original source, but instead return a new sequence which will return the appropriate data
- Query expressions are based on compiler translations of patterns; you don't need to implement any more of the pattern than the relevant query expression requires
- Iterator blocks are great for implementing deferred execution...
- ... but make eager argument validation a pain

Code download

Linq-To-Objects-2.zip

Many people have asked for a source repository for the project, and that makes sense. I'm putting it together a source repository for it now; it's likely to be done before I post the next part.

Back to the table of contents.

Part 3 - Select (and a rename...)

It's been a long time since I wrote part 1 and part 2 of this blog series, but hopefully things will move a bit more quickly now.

The main step forward is that the project now has a source repository on Google Code instead of just being a zip file on each blog post. I had to give the project a title at that point, and I've chosen Edulinq, hopefully for obvious reasons. I've changed the namespaces etc in the code, and the blog tag for the series is now Edulinq too. Anyway, enough of the preamble... let's get on with reimplementing LINQ, this time with the Select operator.

What is it?

Like Where, Select has two overloads:

```
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector)

public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, int, TResult> selector)
```

Again, they both operate the same way - but the second overload allows the index into the sequence to be used as part of the projection.

Simple stuff first: the method *projects* one sequence to another: the "selector" delegate is applied to each input element in turn, to yield an output element. Behaviour notes, which are exactly the same as Where (to the extent that I cut and paste these from the previous blog post, and just tweaked them):

- The input sequence is not modified in any way.
- The method uses *deferred execution* - until you start trying to fetch items from the output sequence, it won't start fetching items from the input sequence.
- Despite deferred execution, it will validate that the parameters aren't null immediately.
- It *streams* its results: it only ever needs to look at one result at a time.
- It will iterate over the input sequence exactly once each time you iterate over the output sequence.
- The "selector" function is called exactly once per yielded value.

- Disposing of an iterator over the output sequence will dispose of the corresponding iterator over the input sequence.

What are we going to test?

The tests are very much like those for Where - except that in cases where we tested the filtering aspect of Where, we're now testing the projection aspect of Select.

There are a few tests of some interest. Firstly, you can tell that the method is generic with 2 type parameters instead of 1 - it has type parameters of TSource and TResult. They're fairly self-explanatory, but it means it's worth having a test for the case where the type arguments are different - such as converting an int to a string:

```
[Test]
public void SimpleProjectionToDifferentType()
{
    int[] source = { 1, 5, 2 };
    var result = source.Select(x => x.ToString());
    result.AssertSequenceEqual("1", "5", "2");
}
```

Secondly, I have a test that shows what sort of bizarre situations you can get into if you include side effects in your query. We could have done this with Where as well of course, but it's clearer with Select:

```
[Test]
public void SideEffectsInProjection()
{
    int[] source = new int[3]; // Actual values won't be relevant
    int count = 0;
    var query = source.Select(x => count++);
    query.AssertSequenceEqual(0, 1, 2);
    query.AssertSequenceEqual(3, 4, 5);
    count = 10;
    query.AssertSequenceEqual(10, 11, 12);
}
```

Notice how we're only calling Select once, but the results of iterating over the results change each time - because the "count" variable has been captured, and is being modified within the projection. Please don't do things like this.

Thirdly, we can now write query expressions which include both "select" and "where" clauses:

```
[Test]
```

```
public void WhereAndSelect()
{
    int[] source = { 1, 3, 4, 2, 8, 1 };
    var result = from x in source
                 where x < 4
                 select x * 2;
    result.AssertSequenceEqual(2, 6, 4, 2);
}
```

There's nothing mind-blowing about any of this, of course - hopefully if you've used LINQ to Objects at all, this should all feel *very* comfortable and familiar.

Let's implement it!

Surprise surprise, we go about implementing Select in much the same way as Where. Again, I simply copied the implementation file and tweaked it a little - the two methods really are that similar. In particular:

- We're using iterator blocks to make it easy to return sequences
- The semantics of iterator blocks mean that we have to separate the argument validation from the real work. (Since I wrote the previous post, I've learned that VB11 will have anonymous iterators, which will avoid this problem. Sigh. It just feels *wrong* to envy VB users, but I'll learn to live with it.)
- We're using foreach within the iterator blocks to make sure that we dispose of the input sequence iterator appropriately - so long as our *output* sequence iterator is disposed or we run out of input elements, of course.

I'll skip straight to the code, as it's all so similar to Where. It's also not worth showing you the version with an index - because it's such a trivial difference.

```
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    if (selector == null)
    {
        throw new ArgumentNullException("selector");
    }
    return SelectImpl(source, selector);
}

private static IEnumerable<TResult> SelectImpl<TSource, TResult>(
    this IEnumerable<TSource> source,
```

```
    Func<TSource, TResult> selector)
{
    foreach (TSource item in source)
    {
        yield return selector(item);
    }
}
```

Simple, isn't it? Again, the real "work" method is even shorter than the argument validation.

Conclusion

While I don't generally like boring my readers (which *may* come as a surprise to some of you) this was a pretty humdrum post, I'll admit. I've emphasized "just like Where" several times to the point of tedium very deliberately though - because it makes it abundantly clear that there aren't really as many tricky bits to understand as you might expect.

Something slightly different next time (which I hope will be in the next few days). I'm not quite sure what yet, but there's an awful lot of methods still to choose from...

Back to the table of contents.

Part 4 - Range

This will be a short post, and there'll probably be some more short ones coming up too. I think it makes sense to only cover multiple operators in a single post where they're *really* similar. (Count and LongCount spring to mind.) I'm in your hands though - if you would prefer "chunkier" posts, please say so in the comments.

This post will deal with the Range generation operator.

What is it?

Range only has a single signature:

```
public static IEnumerable<int> Range(
    int start,
    int count)
```

Unlike most of LINQ, this isn't an extension method - it's a plain old static method. It returns an iterable object which will yield "count" integers, starting from "start" and incrementing each time - so a call to Enumerable.Range(6, 3) would yield 6, then 7, then 8.

As it doesn't operate on an input sequence, there's no sense in which it could stream or buffer its input, but:

- The arguments need to be validated eagerly; the count can't be negative, and it can't be such that any element of the range could overflow Int32.
- The values will be yielded lazily - Range should be *cheap*, rather than creating (say) an array of "count" elements and returning that.

How are we going to test it?

Testing a plain static method brings us a new challenge in terms of switching between the "normal" LINQ implementation and the Edulinq one. This is an artefact of the namespaces I'm using - the tests are in Edulinq.Tests, and the implementation is in Edulinq. "Parent" namespaces are *always* considered when the compiler tries to find a type, and they take priority over anything in using directives - even a using directive which tries to explicitly alias a type name.

The (slightly ugly) solution to this that I've chosen is to include a using directive to create an alias which couldn't otherwise be resolved - in this case, RangeClass. The

using directive will either alias RangeClass to System.Linq.Enumerable or Edulinq.Enumerable. The tests then all use RangeClass.Range. I've also changed how I'm switching between implementations - I now have two project configurations, one of which defines the NORMAL_LINQ preprocessor symbol, and the other of which doesn't. The RangeTest class therefore contains:

```
#if NORMAL_LINQ
using RangeClass = System.Linq.Enumerable;
#else
using RangeClass = Edulinq.Enumerable;
#endif
```

There are alternatives to this approach, of course:

- I could move the tests to a different namespace
- I could make the project *references* depend on the configuration... so the "Normal LINQ" configuration wouldn't reference the Edulinq implementation project, and the "Edulinq implementation" configuration wouldn't reference System.Core. I could then just use Enumerable.Range with an appropriate using directive for System.Linq conditional on the NORMAL_LINQ preprocessor directive, as per the other tests.

I like the idea of the second approach, but it means manually tinkering with the test project file - Visual Studio doesn't expose any way of conditionally including a reference. I may do this at a later date... thoughts welcome.

What are we going to test?

There isn't much we can really test for ranges - I only have eight tests, none of which are particularly exciting:

- A simple valid range should look right when tested with AssertSequenceEqual
- The start value should be allowed to be negative
- Range(Int32.MinValue, 0) is an empty range
- Range(Int32.MaxValue, 1) yields just Int32.MaxValue
- The count *can't* be negative
- The count *can* be zero
- start+count-1 can't exceed Int32.MaxValue (so Range(Int32.MaxValue, 2) isn't valid)
- start+count-1 can *be* Int32.MaxValue (so Range(Int32.MaxValue, 1) *is* valid)

The last two are tested with a few different examples each - a large start and a small count, a small start and a large count, and "fairly large" values for both start and

count.

Note that I *don't* have any tests for lazy evaluation - while I *could* test that the returned value doesn't implement any of the other collection interfaces, it would be a little odd to do so. On the other hand, we *do* have tests which have an enormous count - such that anything which really tried to allocate a collection of that size would almost certainly fail...

Let's implement it!

It will surely be no surprise by now that we're going to use a split implementation, with a public method which performs argument validation eagerly and then uses a private method with an iterator block to perform the actual iteration.

Having validated the arguments, we know that we'll never overflow the bounds of Int32, so we can be pretty casual in the main part of the implementation.

```
public static IEnumerable<int> Range(int start, int count)
{
    if (count < 0)
    {
        throw new ArgumentOutOfRangeException("count");
    }
    // Convert everything to long to avoid overflows. There are other ways of
checking
    // for overflow, but this way make the code correct in the most obvious
way.
    if ((long)start + (long)count - 1L > int.MaxValue)
    {
        throw new ArgumentOutOfRangeException("count");
    }
    return RangeImpl(start, count);
}

private static IEnumerable<int> RangeImpl(int start, int count)
{
    for (int i = 0; i < count; i++)
    {
        yield return start + i;
    }
}
```

Just a few points to note here:

- Arguably it's the *combination* of "start" and "count" which is invalid in the second check, rather than just count. It would possibly be nice to allow ArgumentOutOfRangeException (or ArgumentException in general) to blame

multiple arguments rather than just one. However, using "count" here matches the framework implementation.
- There are other ways of performing the second check, and I certainly didn't have to make *all* the operands in the expression longs. However, I think this is the simplest code which is clearly correct based on the documentation. I don't need to think about all kinds of different situations and check that they all work. The arithmetic will clearly be valid when using the Int64 range of values, so I don't need to worry about overflow, and I don't need to consider whether to use a checked or unchecked context.
- There are also other ways of looping in the private iterator block method, but I think this is the simplest. Another obvious and easy alternative is to keep two values, one for the count of yielded values and the other for the next value to yield, and increment them both on each iteration. A more complex approach would be to use just one loop variable - but you can't use "value < start + count" in case the final value is exactly Int32.MaxValue, and you can't use "value <= start + count - 1" in case the arguments are (int.MinValue, 0). Rather than consider all the border cases, I've gone for an obviously-correct solution. If you really, really cared about the performance of Range, you'd want to investigate various other options.

Prior to writing up this post, I didn't have good tests for Range(Int32.MaxValue, 1) and Range(Int32.MinValue, 0)... but as they could easily go wrong as mentioned above, I've now included them. I find it interesting how considering alternative implementations suggests extra tests.

Conclusion

"Range" was a useful method to implement in order to test some other operators - "Count" in particular. Now that I've started on the non-extension methods though, I might as well do the other two (Empty and Repeat). I've already implemented "Empty", and will hopefully be able to write it up today. "Repeat" shouldn't take much longer, and then we can move on to "Count" and "LongCount".

I think this code is a good example of situations where it's worth writing "dumb" code which looks like the documentation, rather than trying to write possibly shorter, possibly slightly more efficient code which is harder to think about. No doubt there'll be more of that in later posts...

Back to the table of contents.

Part 5 - Empty

Continuing with the non-extension methods, it's time for possibly the simplest LINQ operator around: "Empty".

What is it?

"Empty" is a generic, static method with just a single signature and no parameters:

```
public static IEnumerable<TResult> Empty<TResult>()
```

It returns an empty sequence of the appropriate type. That's all it does.

There's only one bit of interesting behaviour: Empty is *documented* to cache an empty sequence. In other words, it returns a reference to the same empty sequence every time you call it (for the same type argument, of course).

What are we going to test?

There are really only two things we *can* test here:

- The returned sequence is empty
- The returned sequence is cached on a per type argument basis

I'm using the same approach as for Range to call the static method, but this time with an alias of EmptyClass. Here are the tests:

```
[Test]
public void EmptyContainsNoElements()
{
    using (var empty = EmptyClass.Empty<int>().GetEnumerator())
    {
        Assert.IsFalse(empty.MoveNext());
    }
}

[Test]
public void EmptyIsASingletonPerElementType()
{
    Assert.AreSame(EmptyClass.Empty<int>(), EmptyClass.Empty<int>());
    Assert.AreSame(EmptyClass.Empty<long>(), EmptyClass.Empty<long>());
    Assert.AreSame(EmptyClass.Empty<string>(), EmptyClass.Empty<string>());
    Assert.AreSame(EmptyClass.Empty<object>(), EmptyClass.Empty<object>());
```

```
    Assert.AreNotSame(EmptyClass.Empty<long>(), EmptyClass.Empty<int>());
    Assert.AreNotSame(EmptyClass.Empty<string>(),
EmptyClass.Empty<object>());
}
```

Of course, that doesn't verify that the cache isn't per-thread, or something like that... but it'll do.

Let's implement it!

The implementation is actually slightly more interesting than the description so far may suggest. If it weren't for the caching aspect, we *could* just implement it like this:

```
// Doesn't cache the empty sequence
public static IEnumerable<TResult> Empty<TResult>()
{
    yield break;
}
```

... but we want to obey the (somewhat vaguely) documented caching aspect too. It's not really hard, in the end. There's a very handy fact that we can use: **empty arrays are immutable**. Arrays always have a fixed size, but normally there's no way of making an array read-only... you can always change the value of any element. But an empty array doesn't *have* any elements, so there's nothing to change. So, we can reuse the same array over and over again, returning it directly to the caller... but only if we have an empty array of the right type.

At this point you *may* be expecting a `Dictionary<Type, Array>` or something similar... but there's another useful trick we can take advantage of. If you need a per-type cache and the type is being specific as a type argument, you can use static variables in a generic class, because each constructed type will have a distinct set of static variables.

Unfortunately, Empty is a generic method rather than a non-generic method in a generic type... so we've got to create a separate generic type to act as our cache for the empty array. That's easy to do though, and the CLR takes care of initializing the type in a thread-safe way, too. So our final implementation looks like this:

```
public static IEnumerable<TResult> Empty<TResult>()
{
    return EmptyHolder<TResult>.Array;
}
```

```
private static class EmptyHolder<T>
{
    internal static readonly T[] Array = new T[0];
}
```

That obeys all the caching we need, and is really simple in terms of lines of code... but it *does* mean you need to understand how generics work in .NET reasonably well. In some ways this is the opposite of the situation in the previous post - this is a sneaky implementation instead of the slower but arguably simpler dictionary-based one. In this case, I'm happy with the trade-off, because once you *do* understand how generic types and static variables work, this *is* simple code. It's a case where simplicity is in the eye of the beholder.

Conclusion

So, that's Empty. The next operator - Repeat - is likely to be even simpler, although it'll have to be another split implementation...

Addendum

Due to the minor revolt over returning an array (which I still think is fine), here's an alternative implementation:

```
public static IEnumerable<TResult> Empty<TResult>()
{
    return EmptyEnumerable<TResult>.Instance;
}

#if AVOID_RETURNING_ARRAYS
private class EmptyEnumerable<T> : IEnumerable<T>, IEnumerator<T>
{
    internal static IEnumerable<T> Instance = new EmptyEnumerable<T>();

    // Prevent construction elsewhere
    private EmptyEnumerable()
    {
    }

    public IEnumerator<T> GetEnumerator()
    {
        return this;
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return this;
    }
```

```
    public T Current
    {
        get { throw new InvalidOperationException(); }
    }

    object IEnumerator.Current
    {
        get { throw new InvalidOperationException(); }
    }

    public void Dispose()
    {
        // No-op
    }

    public bool MoveNext()
    {
        return false; // There's never a next entry
    }

    public void Reset()
    {
        // No-op
    }
}

#else
private static class EmptyEnumerable<T>
{
    internal static readonly T[] Instance = new T[0];
}
#endif
```

Hopefully now everyone can build a version they're happy with :)

Back to the table of contents.

Part 6 - Repeat

A trivial method next, with even less to talk about than "Empty"... "Repeat". This blog post is merely a matter of completeness.

What is it?

"Repeat" is a static, generic non-extension method with a single overload:

```
public static IEnumerable<TResult> Repeat<TResult>(
    TResult element,
    int count)
```

It simple returns a sequence which contains the specified element, repeated "count" times. The only argument validation is that "count" has to be non-negative.

What are we going to test?

There's really not a lot to test here. I've thought of 4 different scenarios:

- A vanilla "repeat a string 3 times" sequence
- An empty sequence (repeat an element 0 times)
- A sequence containing null values (just to prove that "element" can be null)
- A negative count to prove that argument validation occurs, and does so eagerly.

None of this is remotely exciting, I'm afraid.

Let's implement it!

Just about the only thing we could do wrong here is to put the argument validation directly in an iterator block... and we've implemented the "split method" pattern so many times already that we wouldn't fall into that trap. So, here's the code in all its tedious lack of glory:

```
public static IEnumerable<TResult> Repeat<TResult>(TResult element, int
count)
{
    if (count < 0)
    {
        throw new ArgumentOutOfRangeException("count");
```

```
    }
    return RepeatImpl(element, count);
}

private static IEnumerable<TResult> RepeatImpl<TResult>(TResult element, int
count)
{
    for (int i = 0; i < count; i++)
    {
        yield return element;
    }
}
```

That's it. Um, interesting points to note... none.

Conclusion

There's no sense in dragging this out. That's the lot. Next up, Count and LongCount - which actually *do* have a couple of interesting points.

Back to the table of contents.

Part 7 - Count and LongCount

Today's post covers two operators in one, because they're so incredibly similar... to the point cut and paste of implementation, merely changing the name, return type, and a couple of variables.

What are they?

Count and LongCount each have two overloads: one with a predicate, and one without. Here are all four signatures:

```
public static int Count<TSource>(
    this IEnumerable<TSource> source)

public static int Count<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)

public static long LongCount<TSource>(
    this IEnumerable<TSource> source)

public static long LongCount<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
```

As you can see, the LongCount signatures are identical to Count except in terms of their return types, which are long (Int64) instead of int (Int32).

The overloads without a predicate parameter simply return the number of items in the source collection; the ones with a predicate return the number of items for which that predicate returns true.

Interesting aspects of the behaviour:

- These are all extension methods on IEnumerable<T> - you might argue that for the versions without a predicate, it would have been better to extend the non-generic IEnumerable, as nothing actually requires the element type.
- Where there's no predicate, Count is optimized for ICollection<T> and (in .NET 4) ICollection - both of which have Count properties which are expected to be faster than iterating over the entire collection. LongCount is *not* optimized in the same way in the .NET implementation - I'll discuss this in the implementation section.
- No optimization is performed in the overloads with predicates, as basically

there's no way of telling how many items will "pass" the predicate without testing them.
- All methods use *immediate execution* - nothing is deferred. (If you think about it, there's nothing which *can* be deferred here, when we're just returning an int or a long.)
- All arguments are validated simply by testing they're non-null
- Both methods should throw OverflowException when given a collection with more items than they can return the count of... though this is a considerably larger number in the case of LongCount than Count, of course.

What are we going to test?

In some senses, there are only two "success" tests involved here: one without a predicate and one with. Those are easy enough to deal with, but we also want to exercise the optimized paths. That's actually trickier than it might sound, as we want to test four situations:

- A source which implements both ICollection<T> and ICollection (easy: use List<T>)
- A source which implements ICollection<T> but not ICollection (reasonably easy, after a little work finding a suitable type: use HashSet<T>)
- A source which implements ICollection but not ICollection<T> *but still implements IEnumerable<T>* (so that we can extend it) - tricky...
- A source which doesn't implement ICollection or ICollection<T> (easy: use Enumerable.Range which we've already implemented)

The third bullet is the nasty one. Obviously there are plenty of implementations of ICollection but not ICollection<T> (e.g. ArrayList) but because it doesn't implement IEnumerable<T>, we can't call the Count extension method on it. In the end I wrote my own SemiGenericCollection class.

Once we've got sources for all those tests, we need to decide what we're actually testing about them. Arguably we *should* test that the result is optimized, for example by checking that we never really enumerate the collection. That would require writing custom collections with GetEnumerator() methods which threw exceptions, but still returned a count from the Count property. I haven't gone this far, but it's another step we certainly *could* take.

For the overloads which take predicates, we don't need to worry about the various collection interfaces as we're not optimizing anyway.

The failure cases for null arguments are very simple, but there's one other case to consider: overflow. For Count, I've implemented a test case to verify the overflow

behaviour. Unfortunately we can't run this test in the Edulinq implementation yet, as it requires Enumerable.Concat, but here it is for the record anyway:

```
[Test]
[Ignore("Takes an enormous amount of time!")]
public void Overflow()
{
    var largeSequence = Enumerable.Range(0, int.MaxValue)
                                  .Concat(Enumerable.Range(0, 1));
    Assert.Throws<OverflowException>(() => largeSequence.Count());
}
```

This guards against a bad implementation which overflows by simply wrapping the counter round to Int32.MinValue instead of throwing an exception.

As you can see, this test will be disabled even when it's uncommented after we implement Concat, as it requires counting up to 2 billion - not great for a quick set of unit tests. Even that isn't too bad, however, compared with the equivalent in LongCount which would have to count 2^63 items. Generating such a sequence isn't difficult, but iterating over it all would take a *very* long time. We also need an equivalent test for the overload with a predicate - something I neglected until writing up this blog post, and finding a bug in the implementation as I did so :)

For LongCount, I merely have an equivalent test to the above which checks that the same sequence *can* have its length expressed as a long value.

Let's implement them!

We'll look at the overload which *does* have a predicate first - as it's actually simpler:

```
public static int Count<TSource>(this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    if (predicate == null)
    {
        throw new ArgumentNullException("predicate");
    }

    // No way of optimizing this
    checked
    {
        int count = 0;
        foreach (TSource item in source)
```

```
        {
            if (predicate(item))
            {
                count++;
            }
        }
        return count;
    }
}
```

Note that this time we're not using an iterator block (we're not returning a sequence), so we don't need to split the implementation into two different methods just to get eager argument validation.

After the argument validation, the main part of the method is reasonably simple, with one twist: we're performing the whole iteration within a "checked" context. This means that if the increment of count overflows, it will throw an OverflowException instead of wrapping round to a negative number. There are some other alternatives here:

- We could have made just the increment statement checked instead of the whole second part of the method
- We could have explicitly tested for `count == int.MaxValue` before incrementing, and thrown an exception in that case
- We could just build the whole assembly in a checked context

I think it's useful for this section of code to be explicitly checked - it makes it obvious that it really is a requirement for general correctness. You may well prefer to make only the increment operation checked - I personally believe that the current approach draws more attention to the checked-ness, but it's definitely a subjective matter. It's also possible that an explicit check could be faster, although I doubt it - I haven't benchmarked either approach.

Other than the predicate-specific parts, all the above code also appears in the optimized Count implementation - so I won't discuss those again. Here's the full method:

```
public static int Count<TSource>(this IEnumerable<TSource> source)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }

    // Optimization for ICollection<T>
    ICollection<TSource> genericCollection = source as ICollection<TSource>;
```

```
    if (genericCollection != null)
    {
        return genericCollection.Count;
    }

    // Optimization for ICollection
    ICollection nonGenericCollection = source as ICollection;
    if (nonGenericCollection != null)
    {
        return nonGenericCollection.Count;
    }

    // Do it the slow way - and make sure we overflow appropriately
    checked
    {
        int count = 0;
        using (var iterator = source.GetEnumerator())
        {
            while (iterator.MoveNext())
            {
                count++;
            }
        }
        return count;
    }
}
```

The only "new" code here is the optimization. There are effectively two equivalent blocks, just testing for different collection interface types, and using whichever one it finds first (if any). I don't know whether the .NET implementation tests for ICollection or ICollection<T> first - I could test it by implementing both interfaces but returning different counts from each, of course, but that's probably overkill. It doesn't really matter for well-behaved collections other than the slight performance difference - we want to test the "most likely" interface first, which I believe is the generic one.

To optimize or not to optimize?

The LongCount implementations are exactly the same as those for Count, except using long instead of int.

Notably, I still use optimizations for ICollection and ICollection<T> - but I don't believe the .NET implementation does so. (It's easy enough to tell by creating a huge list of bytes and comparing the time taken for Count and LongCount.)

There's an argument for using Array.GetLongLength when the source is an array... but I don't *think* the current CLR supports arrays with more than Int32.MaxValue elements anyway, so it's a bit of a non-issue other than for future-proofing. Beyond that, I'm not sure why the .NET implementation isn't optimized. It's not clear what an

ICollection/ICollection<T> implementation is meant to return from its Count property if it has more than Int32.MaxValue elements anyway, to be honest.

Suggestions as to what I *should* have done are welcome... but I should probably point out that LongCount is more likely to be used against Queryable than Enumerable - it's easy to imagine a service representing a collection (such as a database table) which can quickly tell you the count even when it's very large. I would imagine that there are relatively few cases where you have a collection to evaluate in-process where you *really* just want to iterate through the whole lot just to get the count.

Conclusion

These are our first LINQ operators which return scalar values instead of sequences - with the natural consequence that they're simpler to understand in terms of control flow and timing. The methods simply execute - possibly with some optimization - and return their result. Nice and simple. Still, we've seen there can still be a few interesting aspects to consider, including questions around optimization which don't necessarily have a good answer.

Next time, I think I'll implement Concat - mostly so that I can uncomment the overflow tests for Count. That's going back to an operator which returns a sequence, but it's a really simple one...

Back to the table of contents.

Part 8 - Concat

After our quick visit to scalar return types with Count and LongCount, we're back to an operator returning a sequence: Concat.

What is it?

Concat only has a single signature, which makes life simple:

```
public static IEnumerable<TSource> Concat<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second)
```

The return value is simply a sequence containing the elements of the first sequence followed by the elements of the second sequence - the *concatenation* of the two sequences.

I sometimes think it's a pity that there aren't Prepend/Append methods which do the same thing but for a single extra element - this would be quite useful in situations such as having a drop down list of countries with an extra option of "None". It's easy enough to use Concat for this purpose by creating a single-element array, but specific methods would be more readable in my view. MoreLINQ has extra Concat methods for this purpose, but Edulinq is only meant to implement the methods already in LINQ to Objects.

As ever, some notes on the behaviour of Concat:

- Arguments are validated eagerly: they must both be non-null
- The result uses deferred execution: other than validation, the arguments aren't used when the method is first called
- Each sequence is only evaluated when it needs to be... if you stop iterating over the output sequence before the first input has been exhausted, the second input will remain unused

That's basically it.

What are we going to test?

The actual concatenation part of the behaviour is very easy to test in a single example - we could potentially also demonstrate concatenation using empty sequences, but there's no reason to suspect they would fail.

The argument validation is tested in the same way as normal, by calling the method with invalid arguments but not attempting to use the returned query.

Finally, there are a couple of tests to indicate the point at which each input sequence is used. This is achieved using the ThrowingEnumerable we originally used in the Where tests:

```
[Test]
public void FirstSequenceIsntAccessedBeforeFirstUse()
{
    IEnumerable<int> first = new ThrowingEnumerable();
    IEnumerable<int> second = new int[] { 5 };
    // No exception yet...
    var query = first.Concat(second);
    // Still no exception...
    using (var iterator = query.GetEnumerator())
    {
        // Now it will go bang
        Assert.Throws<InvalidOperationException>(() => iterator.MoveNext());
    }
}

[Test]
public void SecondSequenceIsntAccessedBeforeFirstUse()
{
    IEnumerable<int> first = new int[] { 5 };
    IEnumerable<int> second = new ThrowingEnumerable();
    // No exception yet...
    var query = first.Concat(second);
    // Still no exception...
    using (var iterator = query.GetEnumerator())
    {
        // First element is fine...
        Assert.IsTrue(iterator.MoveNext());
        Assert.AreEqual(5, iterator.Current);
        // Now it will go bang, as we move into the second sequence
        Assert.Throws<InvalidOperationException>(() => iterator.MoveNext());
    }
}
```

I haven't written tests to check that iterators are disposed, etc - but each input sequence's iterator should be disposed appropriately. In particular, it's natural for the first sequence's iterator to be disposed before the second sequence is iterated over at all.

Let's impement it!

The implementation is reasonably simple, but it does make me hanker after F#... it's

the normal split between argument validation and iterator block implementation, but each part is really simple:

```
public static IEnumerable<TSource> Concat<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second)
{
    if (first == null)
    {
        throw new ArgumentNullException("first");
    }
    if (second == null)
    {
        throw new ArgumentNullException("second");
    }
    return ConcatImpl(first, second);
}

private static IEnumerable<TSource> ConcatImpl<TSource>(
    IEnumerable<TSource> first,
    IEnumerable<TSource> second)
{
    foreach (TSource item in first)
    {
        yield return item;
    }
    foreach (TSource item in second)
    {
        yield return item;
    }
}
```

It's worth just remembering at this point that this would still have been very annoying to implement without iterator blocks. Not really *difficult* as such, but we'd have had to remember which sequence we were currently iterating over (if any) and so on.

However, using F# we could have made this even simpler with the `yield!` expression, which yields a whole sequence instead of a single item. Admittedly in this case there aren't significant performance benefits to using `yield!` (which there certainly can be in recursive situations) but it would just be more elegant to have the ability to yield an entire sequence in one statement. (Spec# has a similar construct called *nested iterators*, expressed using `yield foreach`.) I'm not going to pretend to know enough about the details of either F# or Spec# to draw more detailed comparisons, but we'll see the pattern of "foreach item in a collection, yield the item" several more times before we're done. Remember that we *can't* extract that into a library method, as the "yield" expression needs special treatment by the C# compiler.

Conclusion

Even when presented with a simple implementation, I can still find room to gripe :) It would be nice to have nested iterators in C#, but to be honest the number of times I find myself frustrated by their absence is pretty small.

Concat is a useful operator, but it's really only a very simple specialization of another operator: SelectMany. After all, Concat just flattens two sequences into one... whereas SelectMany can flatten a whole sequence of sequences, with even more generality available when required. I'll implement SelectMany next, and show a few examples of how other operators can be implemented simply in terms of SelectMany. (We'll see the same sort of ability for operators returning a single value when we implement Aggregate.)

Addendum: avoiding holding onto references unnecessarily

A comment suggested that we should set first to "null" after we've used it. That way, as soon as we've finished iterating over the collection, it may be eligible for garbage collection. That leads to an implementation like this:

```
private static IEnumerable<TSource> ConcatImpl<TSource>(
    IEnumerable<TSource> first,
    IEnumerable<TSource> second)
{
    foreach (TSource item in first)
    {
        yield return item;
    }
    // Avoid hanging onto a reference we don't really need
    first = null;
    foreach (TSource item in second)
    {
        yield return item;
    }
}
```

Now normally I'd say this wouldn't actually help - setting a local variable to null when it's not used in the rest of the method doesn't actually make any difference when the CLR is running in optimized mode, without a debugger attached: the garbage collector only cares about variables whih *might* still be accessed in the rest of the method.

In this case, however, it makes a difference - because this *isn't* a normal local variable. It ends up as an instance variable in the hidden class generated by the C# compiler... and the CLR can't tell that the instance variable will never be used again.

Arguably we *could* remove our only reference to "first" at the start of GetEnumerator. We could write a method of the form:

```
public static T ReturnAndSetToNull<T>(ref T value) where T : class
{
    T tmp = value;
    value = null;
    return tmp;
}
```

and then call it like this:

```
foreach (TSource item in ReturnAndSetToNull(ref first))
```

I would certainly consider that overkill, particularly as it seems very likely that the iterator will still have a reference to the collection itself - but simply setting "first" to null after iterating over it makes sense to me.

I don't *believe* that the "real" LINQ to Objects implementation does this, mind you. (At some point I'll test it with a collection which has a finalizer.)

Back to the table of contents.

Part 9 - SelectMany

The next operator we'll implement is actually the most important in the whole of LINQ. Most (all?) of the other operators returning sequences can be implemented via SelectMany. We'll have a look at that at the end of this post, but let's implement it first.

What is it?

SelectMany has 4 overloads, which look gradually more and more scary:

```
public static IEnumerable<TResult> SelectMany<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, IEnumerable<TResult>> selector)

public static IEnumerable<TResult> SelectMany<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, int, IEnumerable<TResult>> selector)

public static IEnumerable<TResult> SelectMany<TSource, TCollection, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, IEnumerable<TCollection>> collectionSelector,
    Func<TSource, TCollection, TResult> resultSelector)

public static IEnumerable<TResult> SelectMany<TSource, TCollection, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, int, IEnumerable<TCollection>> collectionSelector,
    Func<TSource, TCollection, TResult> resultSelector)
```

These aren't too bad though. Really these are just variations of the same operation, with two "optional" bits.

In every case, we start with an input sequence. We generate a subsequence from each element in the input sequence using a delegate which can optionally take a parameter with the index of the element within the original collection.

Now, we *either* return each element from each subsequence directly, *or* we apply another delegate which takes the original element in the input sequence and the element within the subsequence.

In my experience, uses of the overloads where the original selector delegate uses the index are pretty rare - but the others (the first and the third in the list above) are fairly common. In particular, the C# compiler uses the third overload whenever it comes across a "from" clause in a query expression, other than the very first "from" clause.

It helps to put this into a bit more context. Suppose we have a query expression like this:

```
var query = from file in Directory.GetFiles("logs")
            from line in File.ReadLines(file)
            select Path.GetFileName(file) + ": " + line;
```

That would be converted into a "normal" call like this:

```
var query = Directory.GetFiles("logs")
                     .SelectMany(file => File.ReadLines(file),
                                 (file, line) => Path.GetFileName(file) + ":
" + line);
```

In this case the compiler has used our final "select" clause as the projection; if the query expression had continued with "where" clauses etc, it would have created a projection to just pass along "file" and "line" in an anonymous type. This is probably the most confusing bit of the query translation process, involving *transparent identifiers*. For the moment we'll stick with the simple version above.

So, the SelectMany call above has three arguments really:

- The source, which is a list of strings (the filenames returned from Directory.GetFiles)
- An initial projection which converts from a single filename to a list of the lines of text within that file
- A final projection which converts a (file, line) pair into a single string, just by separating them with ": ".

The result is a single sequence of strings - every line of every log file, prefixed with the filename in which it appeared. So writing out the results of the query might give output like this:

```
test1.log: foo
test1.log: bar
test1.log: baz
test2.log: Second log file
test2.log: Another line from the second log file
```

It can take a little while to get your head round SelectMany - at least it did for me - but it's a really important one to understand.

A few more details of the behaviour before we go into testing:

- The arguments are validated eagerly - everything has to be non-null.
- *Everything* is streamed. So only one element of the input is read at a time, and then a subsequence is produced from that. Only one element is then read from the subsequence at a time, yielding the results as we go, before we move onto the next input element and thus the next subsequence etc.
- Every iterator is closed when it's finished with, just as you'd expect by now.

What are we going to test?

I'm afraid I've become lazy by this point. I can't face writing yet *more* tests for null arguments. I've written a single test for each of the overloads. I found it hard to come up with a clear way of writing the tests, but here's one example, for the most complicated overload:

```
[Test]
public void FlattenWithProjectionAndIndex()
{
    int[] numbers = { 3, 5, 20, 15 };
    var query = numbers.SelectMany((x, index) => (x +
index).ToString().ToCharArray(),
                                   (x, c) => x + ": " + c);
    // 3 => "3: 3"
    // 5 => "5: 6"
    // 20 => "20: 2", "20: 2"
    // 15 => "15: 1", "15: 8"
    query.AssertSequenceEqual("3: 3", "5: 6", "20: 2", "20: 2", "15: 1", "15:
8");
}
```

So, to give a bit more explanation to this:

- Each number is summed with its index (3+0, 5+1, 20+2, 15+3)
- Each sum is turned into a string, and then converted into a char array. (We don't really need to ToCharArray call as string implements IEnumerable<char> already, but I thought it made it clearer.)
- We combine each subsequence character with the original element it came from, in the form: "original value: subsequence character"

The comment shows the eventual results from each input, and the final test shows the complete result sequence.

Clear as mud? Hopefully it's not to bad when you look at each step in turn. Okay, now let's make it pass...

Let's implement it!

We *could* implement the first three overloads in terms of calls to the final one - or more likely, a single "Impl" method without argument validation, called by all four public methods. For example, the simplest method *could* be implemented like this:

```
public static IEnumerable<TResult> SelectMany<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, IEnumerable<TResult>> selector)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    if (selector == null)
    {
        throw new ArgumentNullException("selector");
    }
    return SelectManyImpl(source,
                          (value, index) => selector(value),
                          (originalElement, subsequenceElement) =>
subsequenceElement);
}
```

However, I've decided to implement each of the methods separately - splitting them into the public extension method and a "SelectManyImpl" method with the same signature each time. I think that would make it simpler to step through the code if there were ever any problems... and it allows us to see the differences between the simplest and most complicated versions, too:

```
// Simplest overload
private static IEnumerable<TResult> SelectManyImpl<TSource, TResult>(
    IEnumerable<TSource> source,
    Func<TSource, IEnumerable<TResult>> selector)
{
    foreach (TSource item in source)
    {
        foreach (TResult result in selector(item))
        {
            yield return result;
        }
    }
}

// Most complicated overload:
// - Original projection takes index as well as value
// - There's a second projection for each original/subsequence element pair
private static IEnumerable<TResult> SelectManyImpl<TSource, TCollection,
TResult>(
```

```
    IEnumerable<TSource> source,
    Func<TSource, int, IEnumerable<TCollection>> collectionSelector,
    Func<TSource, TCollection, TResult> resultSelector)
{
    int index = 0;
    foreach (TSource item in source)
    {
        foreach (TCollection collectionItem in collectionSelector(item,
index++))
        {
            yield return resultSelector(item, collectionItem);
        }
    }
}
```

The correspondence between the two methods is pretty clear... but I find it helpful to *have* the first form, so that if I ever get confused about the fundamental point of SelectMany, it's really easy to understand it based on the simple overload. It's then not too big a jump to apply the two extra "optional" complications, and end up with the final method. The simple overload acts as a conceptual stepping stone, in a way.

Two minor points to note:

- The first method could have been implemented with a "yield foreach selector(item)" if such an expression existed in C#. Using a similar construct in the more complicated form would be harder, and involve another call to Select, I suspect... probably more hassle than it would be worth.
- I'm not explicitly using a "checked" block in the second form, even though "index" could overflow. I haven't looked to see what the BCL does in this situation, to be honest - I think it unlikely that it will come up. For consistency I should probably use a checked block on *every* method which uses an index like this... or just turn arithmetic checking on for the whole assembly.

Reimplementing operators using SelectMany

I mentioned early on in this post that many of the LINQ operators can be implemented via SelectMany. Just as a quick example of this, here are alternative implementations of Select, Where and Concat:

```
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
```

```
    if (selector == null)
    {
        throw new ArgumentNullException("selector");
    }
    return source.SelectMany(x => Enumerable.Repeat(selector(x), 1));
}

public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    if (predicate == null)
    {
        throw new ArgumentNullException("predicate");
    }
    return source.SelectMany(x => Enumerable.Repeat(x, predicate(x) ? 1 :
0));
}

public static IEnumerable<TSource> Concat<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second)
{
    if (first == null)
    {
        throw new ArgumentNullException("first");
    }
    if (second == null)
    {
        throw new ArgumentNullException("second");
    }
    return new[] { first, second }.SelectMany(x => x);
}
```

Select and Where use Enumerable.Repeat as a convenient way of creating a sequence with either a single element or none. You could alternatively create a new array instead. Concat just uses an array directly: if you think of SelectMany in terms of its flattening operation, Concat is a really natural fit. I suspect that Empty and Repeat are probably feasible with recursion, although the performance would become absolutely horrible.

Currently the above implementations are in the code using conditional compilation. If this becomes a popular thing for me to implement, I might consider breaking it into a separate project. Let me know what you think - my gut feeling is that we won't actually gain much more insight than the above methods give us... just showing how flexible SelectMany is.

SelectMany is also important in a theoretical way, in that it's what provides the *monadic* nature of LINQ. It's the "Bind" operation in the LINQ monad. I don't intend to say any more than that on the topic - read Wes Dyer's blog post for more details, or just search for "bind monad SelectMany" for plenty of posts from people smarter than myself.

Conclusion

SelectMany is one of LINQ's fundamental operations, and at first sight it's a fearsome beast. As soon as you understand that the basic operation is a flattening projection just with a couple of optional twiddles, it's easily tamed.

Next up I'll implement All and Any - which are nice and easy to describe by comparison.

Back to the table of contents.

Part 10 - Any and All

Another day, another blog post. I should emphasize that this rate of posting is likely to be short-lived... although if I get into the habit of writing a post on the morning commute when I go back to work after the Christmas holidays, I could keep ploughing through until we're done.

Anyway, today we have a pair of operators: Any and All.

What are they?

"Any" has two overloads; there's only one for "All":

```
public static bool Any<TSource>(
    this IEnumerable<TSource> source)

public static bool Any<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)

public static bool All<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
```

The names really are fairly self-explanatory:

- "Any" without a predicate returns whether there are *any* elements in the input sequence
- "Any" with a predicate returns whether *any* elements in the input sequence match the predicate
- "All" returns whether *all* the elements in the input sequence match the given predicate

Both operators use *immediate execution* - they don't return until they've got the answer, basically.

Importantly, "All" has to read through the entire input sequence to return true, but can return as soon as it's found a non-matching element; "Any" can return true as soon as it's found a matching element, but has to iterate over the entire input sequence in order to return false. This gives rise to one very simple LINQ performance tip: it's almost never a good idea to use a query like

```
// Don't use this
if (query.Count() != 0)
```

That has to iterate over *all* the results in the query... when you really only care whether or not there are *any* results. So use "Any" instead:

```
// Use this instead
if (query.Any())
```

If this is part of a bigger LINQ to SQL query, it may not make a difference - but in LINQ to Objects it can certainly be a huge boon.

Anyway, let's get on to testing the three methods...

What are we going to test?

Feeling virtuous tonight, I've even tested argument validation again... although it's easy to get that right here, as we're using immediate execution.

Beyond that, I've tested a few scenarios:

- An empty sequence will return false with Any, but true with All. (Whatever the predicate is for All, there are no elements which fail it.)
- A sequence with any elements at all will make the predicate-less Any return true.
- If all the elements *don't* match the predicate, both Any and All return false.
- If *some* elements match the predicate, Any will return true but All will return false.
- If *all* elements match the predicate, All will return true.

Those are all straightforward, so I won't give the code. One final test is interesting though: we prove that Any returns as soon as it's got the result by giving it a query which will throw an exception if it's iterated over completely. The easiest way of doing this is to start out with a sequence of integers including 0, and then use Select with a projection which divides some constant value by each element. In this test case, I've given it a value which will match the predicate *before* the value which will cause the exception to be thrown:

```
[Test]
public void SequenceIsNotEvaluatedAfterFirstMatch()
{
    int[] src = { 10, 2, 0, 3 };
    var query = src.Select(x => 10 / x);
```

```
    // This will finish at the second element (x = 2, so 10/x = 5)
    // It won't evaluate 10/0, which would throw an exception
    Assert.IsTrue(query.Any(y => y > 2));
}
```

There's an equivalent test for All, where a *non-matching* element occurs before the exceptional one.

So, with all the tests written, let's get on with the interesting bit:

Let's implement them!

The first thing to note is that all of these *could* be implemented in terms of either Any-with-a-predicate or All. For example, given All, we could implement Any as:

```
public static bool Any<TSource>(
    this IEnumerable<TSource> source)
{
    return source.Any(x => true);
}

public static bool Any<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    if (predicate == null)
    {
        throw new ArgumentNullException("predicate");
    }
    return !source.All(x => !predicate(x));
}
```

It's simplest to implement the predicate-less Any in terms of the predicated one - using a predicate which returns true for any element means that Any will return true for *any* element at all, which is what we want.

The inversions in the call to All take a minute to get your head round, but it's basically De Morgan's law in LINQ form: we effectively invert the predicate to find out if all of the elements *don't* match the original predicate... then return the inverse. Due to the inversion, this still returns early in all the appropriate situations, too.

While we *could* do that, I've actually preferred a straightforward implementation of all of the separate methods:

```
public static bool Any<TSource>(
    this IEnumerable<TSource> source)
```

```
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }

    using (IEnumerator<TSource> iterator = source.GetEnumerator())
    {
        return iterator.MoveNext();
    }
}

public static bool Any<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    if (predicate == null)
    {
        throw new ArgumentNullException("predicate");
    }

    foreach (TSource item in source)
    {
        if (predicate(item))
        {
            return true;
        }
    }
    return false;
}


public static bool All<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    if (predicate == null)
    {
        throw new ArgumentNullException("predicate");
    }

    foreach (TSource item in source)
    {
        if (!predicate(item))
        {
            return false;
```

```
            }
        }
        return true;
    }
```

Aside from anything else, this makes it obvious where the "early out" comes in each case - and also means that any stack traces generated are rather easier to understand. It would be quite odd from a client developer's point of view to call Any but see All in the stack trace, or vice versa.

One interesting point to note is that I don't actually use a foreach loop in Any - although I could, of course. Instead, I just get the iterator and then return whether the very first call to MoveNext indicates that there are any elements. I like the fact that reading this method it's obvious (at least to me) that we really couldn't care less what the *value* of the first element is - because we never ask for it.

Conclusion

Probably the most important lesson here is the advice to use Any (without a predicate) instead of Count when you can. The rest was pretty simple - although it's always fun to see one operator implemented in terms of another.

So, what next? Possibly Single/SingleOrDefault/First/FirstOrDefault/Last/LastOrDefault. I might as well do them all together - partly as they're so similar, and partly to emphasize the differences which *do* exist.

Back to the table of contents.

Part 11 - First/Single/Last and the ...OrDefault versions

Today I've implemented six operators, each with two overloads. At first I expected the implementations to be very similar, but they've all turned out slightly differently...

What are they?

We've got three axes of permutation here: {First, Last, Single}, {with/without OrDefault }, {with/without a predicate}. That gives us these twelve signatures:

```
public static TSource First<TSource>(
    this IEnumerable<TSource> source)

public static TSource First<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)

public static TSource FirstOrDefault<TSource>(
    this IEnumerable<TSource> source)

public static TSource FirstOrDefault<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)

public static TSource Last<TSource>(
    this IEnumerable<TSource> source)

public static TSource Last<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)

public static TSource LastOrDefault<TSource>(
    this IEnumerable<TSource> source)

public static TSource LastOrDefault<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)

public static TSource Single<TSource>(
    this IEnumerable<TSource> source)

public static TSource Single<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)

public static TSource SingleOrDefault<TSource>(
```

```
    this IEnumerable<TSource> source)

public static TSource SingleOrDefault<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
```

The shared behaviour is as follows:

- They're all extension methods with a single generic type argument
- They're all implemented with *immediate execution*
- They all validate that their parameters are non-null
- The overloads with a predicate are equivalent to calling source.Where(predicate).SameOperator() - in other words, they just add a filter before applying the operator.

With those rules applied, we simply need to consider three possibilities for each operator: what happens if the source sequence is empty, contains a single element, or contains multiple elements. (This is after applying the filter if a predicate is specified, of course.) We can draw the results up in a simple table:

Operator	Empty sequence	Single element	Multiple elements
First	Throws exception	Returns element	Returns first element
FirstOrDefault	Returns default(TSource)	Returns element	Returns first element
Last	Throws exception	Returns element	Returns last element
LastOrDefault	Returns default(TSource)	Returns element	Returns last element
Single	Throws exception	Returns element	Throws exception
SingleOrDefault	Returns default(TSource)	Returns element	Throws exception

As you can see, for an input sequence with a single element, the results are remarkably uniform :) Likewise, for an empty input sequence, any operator without "OrDefault" throws an exception (InvalidOperationException, in fact) and any operator with "OrDefault" returns the default value for the element type (null for reference types, 0 for int etc). The operators really differ if the (potentially filtered) input sequence contains multiple elements - First and Last do the obvious thing, and Single throws an exception. It's worth noting that SingleOrDefault *also* throws an exception - it's not like it's saying, "If the sequence is a single element, return it - otherwise return the default value." If you *want* an operator which handles multiple elements, you should be using First or Last, with the "OrDefault" version if the sequence can legitimately have no elements. Note that if you *do* use an "OrDefault" operator, the result is exactly the same for an empty input sequence as for an input sequence containing exactly one element which is the default value. (I'll be looking at

the DefaultIfEmpty operator next.)

Now we know what the operators do, let's test them.

What are we going to test?

This morning I tweeted that I had written 72 tests before writing any implementation. In fact I ended up with 80, for reasons we'll come to in a minute. For each operator, I tested the following 12 cases:

- Null source (predicate-less overload)
- Null source (predicated overload)
- Null predicate
- Empty sequence, no predicate
- Empty sequence, with a predicate
- Single element sequence, no predicate
- Single element sequence where the element matched the predicate
- Single element sequence where the element didn't match the predicate
- Multiple element sequence, no predicate
- Multiple element
- Multiple element sequence where one element matched the predicate
- Multiple element sequence where multiple elements matched the predicate

These were pretty much cut and paste jobs - I used the same data for each test against each operator, and just changed the expected results.

There are two extra tests for each of First and FirstOrDefault, and two for each of Last and LastOrDefault:

- First/FirstOrDefault should return as soon as they've seen the first element, when there's no predicate; they shouldn't iterate over the rest of the sequence
- First/FirstOrDefault should return as soon as they've seen the first *matching* element, when there is a predicate
- Last/LastOrDefault are optimized for the case where the source implements IList<T> and there's no predicate: it uses Count and the indexer to access the final element
- Last/LastOrDefault is *not* optimized for the case where the source implements IList<T> but there is a predicate: it iterates through the entire sequence

The last two tests involved writing a new collection called NonEnumerableList which implements IList<T> by delegating everything to a backing List<T>, except for GetEnumerator() (both the generic and nongeneric forms) which simply throws a NotSupportedException. That should be a handy one for testing optimizations in the future. I'll discuss the optimization for Last when we get there.

Let's implement them!

These operators were more interesting to implement than I'd expect, so I'm actually going to show all twelve methods. It was rarely just a matter of cutting and pasting, other than for the argument validation.

Of course, if we chose to implement the predicated versions using "Where" and the non-predicated form, and the "OrDefault" versions by using "DefaultIfEmpty" followed by the non-defaulting version, we would only have had the three non-predicated, non-defaulting versions to deal with... but as I've said before, there are some virtues to implementing each operator separately.

For the sake of avoiding fluff, I've removed the argument validation from each method - but obviously it's there in the real code. Let's start with First:

```
public static TSource First<TSource>(
    this IEnumerable<TSource> source)
{
    // Argument validation elided
    using (IEnumerator<TSource> iterator = source.GetEnumerator())
    {
        if (iterator.MoveNext())
        {
            return iterator.Current;
        }
        throw new InvalidOperationException("Sequence was empty");
    }
}

public static TSource First<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    // Argument validation elided
    foreach (TSource item in source)
    {
        if (predicate(item))
        {
            return item;
        }
    }
    throw new InvalidOperationException("No items matched the predicate");
}
```

These look surprisingly different - and that was actually a deliberate decision. I could easily have implemented the predicate-less version with a foreach loop as well, just returning unconditionally from its body. However, I chose to emphasize the fact that

we're *not* looping in First: we simply move to the first element if we can, and return it or throw an exception. There's no *hint* that we might ever call MoveNext again. In the predicated form, of course, we have to keep looping until we find a matching value - only throwing the exception when we've exhausted all possibilities.

Now let's see how it looks when we return a default for empty sequences:

```
public static TSource FirstOrDefault<TSource>(
    this IEnumerable<TSource> source)
{
    // Argument validation elided
    using (IEnumerator<TSource> iterator = source.GetEnumerator())
    {
        return iterator.MoveNext() ? iterator.Current : default(TSource);
    }
}

public static TSource FirstOrDefault<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    // Argument validation elided
    foreach (TSource item in source)
    {
        if (predicate(item))
        {
            return item;
        }
    }
    return default(TSource);
}
```

Here the predicated form looks very similar to First, but the predicate-less one is slightly different: instead of using an if block (which would be a perfectly valid approach, of course) I've used the conditional operator. We're *going* to return something, whether we manage to move to the first element or not. Arguably it would be nice if the conditional operator allowed the second or third operands to be "throw" expressions, taking the overall type of the expression from the other result operand... but it's no great hardship.

Next up we'll implement Single, which is actually closer to First than Last is, in some ways:

```
public static TSource Single<TSource>(
    this IEnumerable<TSource> source)
{
    // Argument validation elided
    using (IEnumerator<TSource> iterator = source.GetEnumerator())
```

```
    {
        if (!iterator.MoveNext())
        {
            throw new InvalidOperationException("Sequence was empty");
        }
        TSource ret = iterator.Current;
        if (iterator.MoveNext())
        {
            throw new InvalidOperationException("Sequence contained multiple
elements");
        }
        return ret;
    }
}

public static TSource Single<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    // Argument validation elided
    TSource ret = default(TSource);
    bool foundAny = false;
    foreach (TSource item in source)
    {
        if (predicate(item))
        {
            if (foundAny)
            {
                throw new InvalidOperationException("Sequence contained
multiple matching elements");
            }
            foundAny = true;
            ret = item;
        }
    }
    if (!foundAny)
    {
        throw new InvalidOperationException("No items matched the
predicate");
    }
    return ret;
}
```

This is already significantly more complex than First. The predicate-less version starts off the same way, but if we manage to move to the first element, we have to remember that value (as we're *hoping* to return it) and then try to move to the *second* element. This time, if the move succeeds, we have to throw an exception - otherwise we can return our saved value.

The predicated version is even hairier. We still need to remember the first matching value we find, but this time we're looping - so we need to keep track of whether we've

already seen a matching value or not. If we see a second match, we have to throw an exception... and we *also* have to throw an exception if we reach the end without finding any matches at all. Note that although we assign an initial value of default(TSource) to ret, we'll never reach a return statement without assigning a value to it. However, the rules around definite assignment aren't smart enough to cope with this, so we need to provide a "dummy" value to start with... and default(TSource) is really the only value available. There is an alternative approach without using a foreach statement, where you loop until you find the first match, assign it to a local variable which is only declared at that point, followed by a second loop ensuring that there aren't any other matches. I personally think that's a bit more complex, which is why I've just used the foreach here.

The difference when we implement SingleOrDefault isn't quite as pronounced this time though:

```
public static TSource SingleOrDefault<TSource>(
    this IEnumerable<TSource> source)
{
    // Argument validation elided
    using (IEnumerator<TSource> iterator = source.GetEnumerator())
    {
        if (!iterator.MoveNext())
        {
            return default(TSource);
        }
        TSource ret = iterator.Current;
        if (iterator.MoveNext())
        {
            throw new InvalidOperationException("Sequence contained multiple
elements");
        }
        return ret;
    }
}

public static TSource SingleOrDefault<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    // Argument validation elided
    TSource ret = default(TSource);
    bool foundAny = false;
    foreach (TSource item in source)
    {
        if (predicate(item))
        {
            if (foundAny)
            {
                throw new InvalidOperationException("Sequence contained
multiple matching elements");
```

```
            }
            foundAny = true;
            ret = item;
        }
    }
    return ret;
}
```

This time we've just replaced a "throw" statement in the predicate-less method with a "return" statement, and removed the test for no matches being found in the predicated method. Here our assignment of default(TSource) to ret really works in our favour - if we don't end up assigning anything else to it, we've already got the right return value!

Next up is Last:

```
public static TSource Last<TSource>(
    this IEnumerable<TSource> source)
{
    // Argument validation elided
    IList<TSource> list = source as IList<TSource>;
    if (list != null)
    {
        if (list.Count == 0)
        {
            throw new InvalidOperationException("Sequence was empty");
        }
        return list[list.Count - 1];
    }

    using (IEnumerator<TSource> iterator = source.GetEnumerator())
    {
        if (!iterator.MoveNext())
        {
            throw new InvalidOperationException("Sequence was empty");
        }
        TSource last = iterator.Current;
        while (iterator.MoveNext())
        {
            last = iterator.Current;
        }
        return last;
    }
}

public static TSource Last<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    // Argument validation elided
    bool foundAny = false;
```

```
    TSource last = default(TSource);
    foreach (TSource item in source)
    {
        if (predicate(item))
        {
            foundAny = true;
            last = item;
        }
    }
    if (!foundAny)
    {
        throw new InvalidOperationException("No items matched the
predicate");
    }
    return last;
}
```

Let's start off with the optimization at the beginning of the predicate-less method. If we find out we're dealing with a list, we can simply fetch the count, and then either throw an exception or return the element at the final index. As an extra bit of optimization I *could* store the count in a local variable - but I'm assuming that the count of an IList<T> is cheap to compute. If there are significant objections to that assumption, I'm happy to change it :) Note that this is another situation where I'm assuming that anything implementing IList<T> will only hold at most Int32.MaxValue items - otherwise the optimization will fail.

If we *don't* follow the optimized path, we simply iterate over the sequence, updating a local variable with the last-known element on every single iteration. This time I've avoided the foreach loop for no particularly good reason - we could easily have had a foundAny variable which was just set to "true" on every iteration, and then tested at the end. In fact, that's exactly the pattern the predicated method takes. Admittedly that decision is forced upon us to some extent - we can't just move once and then take the first value as the "first last-known element", because it might not match the predicate.

There's no optimization for the predicated form of Last. This follows LINQ to Objects, but I don't honestly know the reason for it there. We could easily iterate backwards from the end of the sequence using the indexer on each iteration. One possible reason which makes a certain amount of sense is that when there's a predicate, that predicate could throw an exception for some values - and if we just skipped to the end if the collection implements IList<T>, that would be an observable difference. I'd be interested to know whether or not that *is* the reason - if anyone has any inside information which they can share, I'll update this post.

From here, we only have one more operator to implement - LastOrDefault:

```
public static TSource LastOrDefault<TSource>(
    this IEnumerable<TSource> source)
{
    // Argument validation elided
    IList<TSource> list = source as IList<TSource>;
    if (list != null)
    {
        return list.Count == 0 ? default(TSource) : list[list.Count - 1];
    }

    TSource last = default(TSource);
    foreach (TSource item in source)
    {
        last = item;
    }
    return last;
}

public static TSource LastOrDefault<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    // Argument validation elided
    TSource last = default(TSource);
    foreach (TSource item in source)
    {
        if (predicate(item))
        {
            last = item;
        }
    }
    return last;
}
```

This time, aside from the optimization, the predicated and non-predicated forms look *very* similar... more so than for any of the other operators. In each case, we start with a return value of default(TSource), and iterate over the whole sequence, updating it - only doing so when it matches the predicate, if we've got one.

Conclusion

This was a longer post than I anticipated when I got up this morning, but I hope the slight differences between implementations - and the mystery of the unoptimized predicated "Last/LastOrDefault" operators have made it worth slogging through.

As a contrast - and because I've already mentioned it in this post - I'll implement DefaultIfEmpty next. I reckon I can still do that this evening, if I hurry...

Addendum

It turns out I was missing some tests for Single and SingleOrDefault: what should they do if evaluating the sequence fully throws an exception? It turns out that in LINQ to Objects, the overloads *without* a predicate throw InvalidOperationException as soon as they see a second element, but the overloads *with* a predicate keep iterating even when they've seen a second element matching a predicate. This seems ludicrously inconsistent to me - I've opened a Connect issue about it; we'll see what happens.

Back to the table of contents.

Part 12 - DefaultIfEmpty

After the masses of code required for all the permutations of First/Last/etc, DefaultIfEmpty is a bit of a relief.

What is it?

Even this simple operator has two overloads:

```
public static IEnumerable<TSource> DefaultIfEmpty<TSource>(
    this IEnumerable<TSource> source)

public static IEnumerable<TSource> DefaultIfEmpty<TSource>(
    this IEnumerable<TSource> source,
    TSource defaultValue)
```

The behaviour is very simple to describe:

- If the input sequence is empty, the result sequence has a single element in it, the default value. This is default(TSource) for the overload without an extra parameter, or the specified value otherwise.
- If the input sequence isn't empty, the result sequence is the same as the input sequence
- The source argument must not be null, and this is validated eagerly
- The result sequence itself uses *deferred execution* - the input sequence isn't read at all until the result sequence is read
- The input sequence is streamed; any values read are yielded immediately; no buffering is used

Dead easy.

What are we going to test?

Despite being relatively late in the day, I decided to test for argument validation - and a good job too, as my first attempt failed to split the implementation into an argument validation method and an iterator block method for the real work! It just shows how easy it is to slip up.

Other than that, I can really only see four cases worth testing:

- No default value specified, empty input sequence

- Default value specified, empty input sequence
- No default value specified, non-empty input sequence
- Default value specified, non-empty input sequence

So I have tests for all of those, and that's it. I don't have anything testing for streaming, lazy evaluation etc.

Let's implement it!

Despite my reluctance to implement one operator in terms of another elsewhere, this felt like *such* an obvious case, I figured it would make sense just this once. I even applied DRY to the argument validation aspect. Here's the implementation in all its brief glory:

```
public static IEnumerable<TSource> DefaultIfEmpty<TSource>(
    this IEnumerable<TSource> source)
{
    // This will perform an appropriate test for source being null first.
    return source.DefaultIfEmpty(default(TSource));
}

public static IEnumerable<TSource> DefaultIfEmpty<TSource>(
    this IEnumerable<TSource> source,
    TSource defaultValue)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    return DefaultIfEmptyImpl(source, defaultValue);
}

private static IEnumerable<TSource> DefaultIfEmptyImpl<TSource>(
    IEnumerable<TSource> source,
    TSource defaultValue)
{
    bool foundAny = false;
    foreach (TSource item in source)
    {
        yield return item;
        foundAny = true;
    }
    if (!foundAny)
    {
        yield return defaultValue;
    }
}
```

Of course, now that I've said how easy it was, someone will find a bug...

Aside from the use of default(TSource) to call the more complex overload from the simpler one, the only aspect of any interest is the bottom method. It irks me slightly that we're assigning "true" to "foundAny" on every iteration... but the alternative is fairly unpleasant:

```
private static IEnumerable<TSource> DefaultIfEmptyImpl<TSource>(
    IEnumerable<TSource> source,
    TSource defaultValue)
{
    using (IEnumerator<TSource> iterator = source.GetEnumerator())
    {
        if (!iterator.MoveNext())
        {
            yield return defaultValue;
            yield break; // Like a "return"
        }
        yield return iterator.Current;
        while (iterator.MoveNext())
        {
            yield return iterator.Current;
        }
    }
}
```

This may be slightly more efficient, but it *feels* a little clumsier. We could get rid of the "yield break" by putting the remainder of the method in an "else" block, but I'm not dead keen on that, either. We could use a do/while loop instead of a simple while loop - that would at least remove the repetition of "yield return iterator.Current" but I'm not really a fan of do/while loops. I use them sufficiently rarely that they cause me more mental effort to read than I really like.

If any readers have suggestions which are significantly nicer than either of the above implementations, I'd be interested to hear them. This feels a little inelegant at the moment. It's far from a readability disaster - it's just not quite neat.

Conclusion

Aside from the slight annoyance at the final lack of elegance, there's not much of interest here. However, we could now implement FirstOrDefault/LastOrDefault/SingleOrDefault using DefaultIfEmpty. For example, here's an implementation of FirstOrDefault:

```
public static TSource FirstOrDefault<TSource>(
    this IEnumerable<TSource> source)
{
```

```
    return source.DefaultIfEmpty().First();
}

public static TSource FirstOrDefault<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    // Can't just use source.DefaultIfEmpty().First(predicate)
    return source.Where(predicate).DefaultIfEmpty().First();
}
```

Note the comment in the predicated version - the defaulting has to be the *very last step* after we've applied the predicate... otherwise if we pass in an empty sequence and a predicate which doesn't match with default(TSource), we'll get an exception instead of the default value. The other two ...OrDefault operators could be implemented in the same way, of course. (I haven't done so yet, but the above code is in source control.)

I'm currently unsure what I'll implement next. I'll see whether I get inspired by any particular method in the morning.

Back to the table of contents.

Part 13 - Aggregate

EDIT: I've had to edit this quite a bit now that a second bug was discovered... basically my implementation of the first overload was completely broken :(

Last night's tweet asking for suggestions around which operator to implement next resulted in a win for Aggregate, so here we go.

What is it?

Aggregate has three overloads, effectively allow two defaults:

```
public static TSource Aggregate<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, TSource, TSource> func)

public static TAccumulate Aggregate<TSource, TAccumulate>(
    this IEnumerable<TSource> source,
    TAccumulate seed,
    Func<TAccumulate, TSource, TAccumulate> func)

public static TResult Aggregate<TSource, TAccumulate, TResult>(
    this IEnumerable<TSource> source,
    TAccumulate seed,
    Func<TAccumulate, TSource, TAccumulate> func,
    Func<TAccumulate, TResult> resultSelector)
```

Aggregate is an extension method using immediate execution, returning a single result. The generalised behaviour is as follows:

- Start off with a *seed*. For the first overload, this defaults to the first value of the input sequence. The seed is used as the first *accumulator* value.
- For each item in the list, apply the aggregation function, which takes the current accumulator value and the newly found item, and returns a new accumulator value.
- Once the sequence has been exhausted, optionally apply a final projection to obtain a result. If no projection has been specified, we can imagine that the identity function has been provided.

The signatures make all of this look a bit more complicated because of the various type parameters involved. You can consider all the overloads as dealing with three different types, even though the first two actually have fewer type parameters:

- TSource is the element type of the sequence, always.
- TAccumulate is the type of the accumulator - and thus the seed. For the first overload where no seed is provided, TAccumulate is effectively the same as TSource.
- TResult is the return type when there's a final projection involved. For the first two overloads, TResult is effectively the same as TAccumulate (again, think of a default "identity projection" as being used when nothing else is specified)

In the first overload, which uses the first input element as the seed, an InvalidOperationException is thrown if the input sequence is empty.

What are we going to test?

Obviously the argument validation is reasonably simple to test - source, func and resultSelector can't be null. But there are two different approaches to testing the "success" cases.

We *could* work out exactly when each delegate should be called and with what values - effectively mock every step of the iteration. This would be a bit of a pain, but a very robust way of proceeding.

The alternative approach is just to take some sample data and aggregation function, work out what the result should be, and assert that result. If the result is sufficiently unlikely to be achieved by chance, this is probably good enough - and it's a lot simpler to implement. Here's a sample from the most complicated test, where we have a seed and a final projection:

```
[Test]
public void SeededAggregationWithResultSelector()
{
    int[] source = { 1, 4, 5 };
    int seed = 5;
    Func<int, int, int> func = (current, value) => current * 2 + value;
    Func<int, string> resultSelector = result => result.ToInvariantString();
    // First iteration: 5 * 2 + 1 = 11
    // Second iteration: 11 * 2 + 4 = 26
    // Third iteration: 26 * 2 + 5 = 57
    // Result projection: 57.ToString() = "57"
    Assert.AreEqual("57", source.Aggregate(seed, func, resultSelector));
}
```

Now admittedly I'm not testing this to the absolute full - I'm using the same types for TSource and TAccumulate - but frankly it gives me plenty of confidence that the implementation is correct.

EDIT: My result selector now calls ToInvariantString. It used to just call ToString, but as I've now been persuaded that there are some cultures where that wouldn't give us the right results, I've implemented an extension method which effectively means that x.ToInvariantString() is equivalent to x.ToString(CultureInfo.InvariantCulture) - so we don't need to worry about cultures with different numeric representations etc.

Just for the sake of completeness (I've convinced myself to improve the code while writing this blog post), here's an example which sums integers, but results in a long - so it copes with a result which is bigger than Int32.MaxValue. I haven't bothered with a final projection though:

```
[Test]
public void DifferentSourceAndAccumulatorTypes()
{
    int largeValue = 2000000000;
    int[] source = { largeValue, largeValue, largeValue };
    long sum = source.Aggregate(0L, (acc, value) => acc + value);
    Assert.AreEqual(6000000000L, sum);
    // Just to prove we haven't missed off a zero...
    Assert.IsTrue(sum > int.MaxValue);
}
```

Since I first wrote this post, I've also added tests for empty sequences (where the first overload should throw an exception) and a test which relies on the first overload using the first element of the sequence as the seed, rather than the default value of the input sequence's element type.

Okay, enough about the testing... what about the real code?

Let's implement it!

I'm still feeling my way around when it's a good idea to implement one method by using another, but at the moment my gut feeling is that it's okay to do so when:

- You're implementing one operator by reusing another overload of the same operator; in other words, no unexpected operators will end up in the stack trace of callers
- There are no significant performance penalties for doing so
- The observed behaviour is *exactly* the same - including argument validation
- The code ends up being simpler to understand (obviously)

Contrary to an earlier version of this post, the first overload can't be implemented in terms of the second or third ones, because of its behaviour regarding the seed and empty sequences.

```
public static TSource Aggregate<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, TSource, TSource> func)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    if (func == null)
    {
        throw new ArgumentNullException("func");
    }

    using (IEnumerator<TSource> iterator = source.GetEnumerator())
    {
        if (!iterator.MoveNext())
        {
            throw new InvalidOperationException("Source sequence was empty");
        }
        TSource current = iterator.Current;
        while (iterator.MoveNext())
        {
            current = func(current, iterator.Current);
        }
        return current;
    }
}
```

It still makes sense to share an implementation for the second and third overloads though. There's a choice around whether to implement the second operator in terms of the third (giving it an identity projection) or to implement the third operator in terms of the second (by just calling the second overload and then applying a projection). Obviously applying an unnecessary identity projection has a performance penalty in itself - but it's a *tiny* penalty. So which is more readable? I'm in two minds about this. I like code where various methods call one other "central" method where *all* the real work occurs (suggesting implementing the second overload using the third) but equally I suspect I really think about aggregation in terms of getting the final value of the accumulator... with just a twist in the third overload, of an extra projection. I guess it depends on whether you think of the final projection as part of the general form or an "extra" step.

For the moment, I've gone with the "keep all logic in one place" approach:

```
public static TAccumulate Aggregate<TSource, TAccumulate>(
    this IEnumerable<TSource> source,
    TAccumulate seed,
    Func<TAccumulate, TSource, TAccumulate> func)
{
```

```
    return source.Aggregate(seed, func, x => x);
}

public static TResult Aggregate<TSource, TAccumulate, TResult>(
    this IEnumerable<TSource> source,
    TAccumulate seed,
    Func<TAccumulate, TSource, TAccumulate> func,
    Func<TAccumulate, TResult> resultSelector)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    if (func == null)
    {
        throw new ArgumentNullException("func");
    }
    if (resultSelector == null)
    {
        throw new ArgumentNullException("resultSelector");
    }
    TAccumulate current = seed;
    foreach (TSource item in source)
    {
        current = func(current, item);
    }
    return resultSelector(current);
}
```

The bulk of the "real work" method is argument validation - the actual iteration is almost painfully simple.

Conclusion

The moral of today's story is to read the documentation carefully - sometimes there's unexpected behaviour to implement. I still don't really know *why* this difference in behaviour exists... it feels to me as if the first overload really *should* behave like the second one, just with a default initial seed. EDIT: it seems that you need to read it *really* carefully. You know, every word of it. Otherwise you could make an embarrassing goof in a public blog post. <sigh>

The second moral should really be about the use of Aggregate - it's a *very* generalized operator, and you can implement any number of other operators (Sum, Max, Min, Average etc) using it. In some ways it's the scalar equivalent of SelectMany, just in terms of its diversity. Maybe I'll show some later operators implemented using Aggregate...

Next up, there have been requests for some of the set-based operators - Distinct,

Union, etc - so I'll probably look at those soon.

Back to the table of contents.

Part 14 - Distinct

I'm going to implement the set-based operators next, starting with Distinct.

What is it?

Distinct has two overloads, which differ only in the simplest possible way:

```
public static IEnumerable<TSource> Distinct<TSource>(
    this IEnumerable<TSource> source)

public static IEnumerable<TSource> Distinct<TSource>(
    this IEnumerable<TSource> source,
    IEqualityComparer<TSource> comparer)
```

The point of the operator is straightforward: the result sequence contains the same items as the input sequence, but with the duplicates removed - so an input of { 0, 1, 3, 1, 5 } will give a result sequence of { 0, 1, 3, 5 } - the second occurrence of 1 is ignored.

This time I've checked and double-checked the documentation - and in this case it really *is* appropriate to think of the first overload as just a simplified version of the second. If you don't specify an equality comparer, the default comparer for the type will be used. The same will happen if you pass in a null reference for the comparer. The default equality comparer for a type can be obtained with the handy EqualityComparer<T>.Default property.

Just to recap, an *equality comparer* (represented by the IEqualityComparer<T> interface) is able to do two things:

- Determine the hash code for a single item of type T
- Compare any two items of type T for equality

It doesn't have to give any sort of ordering - that's what IComparer<T> is for, although that *doesn't* have the ability to provide a hash code.

One interesting point about IEqualityComparer<T> is that the GetHashCode() method is *meant* to throw an exception if it's provided with a null argument, but in practice the EqualityComparer<T>.Default implementations appear not to. This leads to an interesting question about Distinct: how should it handle null elements? It's not documented either way, but in reality both the LINQ to Objects implementation

and the simplest way of implementing it ourselves simply throws a NullReferenceException if you use a not-null-safe comparer and have a null element present. Note that the *default* equality comparer for any type (EqualityComparer<T>.Default) *does* cope with nulls.

There are other undocumented aspects of Distinct, too. Both the ordering of the result sequence and the choice of which exact element is returned when there are equal options are unspecified. In the case of ordering, it's *explicitly* unspecified. From the documentation: "The Distinct method returns an unordered sequence that contains no duplicate values." However, there's a natural approach which answers both of these questions. Distinct *is* specified to use deferred execution (so it won't look at the input sequence until you start reading from the output sequence) but it also streams the results to some extent: to return the first element in the result sequence, it only needs to read the first element from the input sequence. Some other operators (such as OrderBy) have to read *all* their data before yielding any results.

When you implement Distinct in a way which only reads as much data as it has to, the answer to the ordering and element choice is easy:

- The result sequence is in the same order as the input sequence
- When there are multiple equal elements, it is the one which occurs earliest in the input sequence which is returned as part of the result sequence.

Remember that it's perfectly possible to have elements which are considered equal under a particular comparer, but are still clearly different when looked at another way. The simplest example of this is case-insensitive string equality. Taking the above rules into account, the distinct sequence returned for { "ABC", "abc", "xyz" } with a case-insensitive comparer is { "ABC", "xyz" }.

What are we going to test?

All of the above :)

All the tests use sequences of strings for clarity, but I'm using four different comparers:

- The default string comparer (which is a case-sensitive ordinal comparer)
- The case-insensitive ordinal comparer
- A comparer which uses object identity (so will treat two equal but distinct strings as different)
- A comparer which explicitly *doesn't* try to cope with null values

The tests assume that the undocumented aspects listed above are implemented with

the rules that I've given. This means they're over-sensitive, in that an implementation of Distinct which matches all the *documented* behaviour but returns elements in a different order would fail the tests. This highlights an interesting aspect of unit testing in general... what exactly are we trying to test? I can think of three options in our case:

- Just the documented behaviour: anything conforming to that, however oddly, should pass
- The LINQ to Objects behaviour: the framework implementation should pass all our tests, and then our implementation should as well
- Our implementation's known (designed) behaviour: we can specify that our implementation will follow particular rules above and beyond the documented contracts

In production projects, these different options are valid in different circumstances, depending on exactly what you're trying to do. At the moment, I don't *have* any known differences in behaviour between LINQ to Objects and Edulinq, although that may well change later in terms of optimizations.

None of the tests themselves are particularly interesting - although I find it interesting that I had to implement a deliberately fragile (but conformant) implementation of IEqualityComparer<T> in order to test Distinct fully.

Let's implement it!

I'm absolutely confident in implementing the overload that doesn't take a custom comparer using the one that does. We have two options for how to specify the custom comparer in the delegating call though - we could pass null or EqualityComparer<T>.Default, as the two are explicitly defined to behave the same way in the second overload. I've chosen to pass in EqualityComparer<T>.Default just for the sake of clarity - it means that anyone reading the first method doesn't need to check the behavior of the second to understand what it will do.

We need to use the "private iterator block method" approach again, so that the arguments can be evaluated eagerly but still let the result sequence use deferred execution. The real work method uses HashSet<T> to keep track of all the elements we've already returned - it takes an IEqualityComparer<T> in its constructor, and the Add method adds an element to the set if there isn't already an equal one, and returns whether or not it *really* had to add anything. All we have to do is iterate over the input sequence, call Add, and yield the item as part of the result sequence if Add returned true. Simple!

```
public static IEnumerable<TSource> Distinct<TSource>(
    this IEnumerable<TSource> source)
```

```
{
    return source.Distinct(EqualityComparer<TSource>.Default);
}

public static IEnumerable<TSource> Distinct<TSource>(
    this IEnumerable<TSource> source,
    IEqualityComparer<TSource> comparer)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    return DistinctImpl(source, comparer ??
EqualityComparer<TSource>.Default);
}

private static IEnumerable<TSource> DistinctImpl<TSource>(
    IEnumerable<TSource> source,
    IEqualityComparer<TSource> comparer)
{
    HashSet<TSource> seenElements = new HashSet<TSource>(comparer);
    foreach (TSource item in source)
    {
        if (seenElements.Add(item))
        {
            yield return item;
        }
    }
}
```

So what about the behaviour with nulls? Well, it seems that HashSet<T> just handles that automatically, if the comparer it uses does. So long as the comparer returns the same hash code each time it's passed null, and considers null and null to be equal, it can be present in the sequence. Without HashSet<T>, we'd have had a much uglier implementation - especially as Dictionary<TKey, TValue> doesn't allow null keys.

Conclusion

I'm frankly bothered by the lack of specificity in the documentation for Distinct. Should you rely on the ordering rules that I've given here? I think that in reality, you're reasonably safe to rely on it - it's the natural result of the most obvious implementation, after all. I wouldn't rely on the same results when using a different LINQ provider, mind you - when fetching the results back from a database, for example, I wouldn't be at all surprised to see the ordering change. And of course, the fact that t documentation explicitly states that the result is unordered should act as a bit of a deterrent from relying on this.

We'll have to make similar decisions for the other set-based operators: Union,

Intersect and Except. And yes, they're very likely to use HashSet<T> too...

Back to the table of contents.

Part 15 - Union

I'm continuing the set-based operators with Union. I may even have a couple of hours tonight - possibly enough to finish off Intersect and Except as well... let's see.

What is it?

Union is another extension method with two overloads; one taking an equality comparer and one just using the default:

```
public static IEnumerable<TSource> Union<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second)

public static IEnumerable<TSource> Union<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second,
    IEqualityComparer<TSource> comparer)
```

This "two overloads, one taking an equality comparer" pattern is familiar from Distinct; we'll see that and Intersect and Except do exactly the same thing.

Simply put, Union returns the union of the two sequences - all items that are in either input sequence. The result sequence has no duplicates in even if one of the input sequences contains duplicates. (I'm using the term "duplicate" here to mean an element which is equal to another according to the equality comparer we're using in the operator.)

Characteristics:

- Union uses deferred execution: argument validation is basically all that happens when the method is first called; it only starts iterating over the input sequences when you iterate over the result sequence
- Neither first nor second can be null; the comparer argument *can* be null, in which case the default equality comparer is used
- The input sequences are only read as and when they're needed; to return the first result element, only the first input element is read

It's worth noting that the documentation for Union specifies a lot more than the Distinct documentation does:

> When the object returned by this method is enumerated, Union enumerates first

and second in that order and yields each element that has not already been yielded.

To me, that actually looks like a *guarantee* of the rules I proposed for Distinct. In particular, it's guaranteeing that the implementation iterates over "first" before "second", and if it's yielding elements as it goes, that guarantees that distinct elements will retain their order from the original input sequences. Whether others would read it in the same way or not, I can't say... input welcome.

What are we going to test?

I've written quite a few tests for Union - possibly more than we really *need*, but they demonstrate a few points of usage. The tests are:

- Arguments are validated eagerly
- Finding the union of two sequences without specifying a comparer; both inputs have duplicate elements, and there's one element in both
- The same test as above but explicitly specifying null as the comparer, to force the default to be used
- The same test as above but using a case-insensitive string comparer
- Taking the union of an empty sequence with a non-empty one
- Taking the union of a non-empty sequence with an empty one
- Taking the union of two empty sequences
- Proving that the first sequence isn't used until we start iterating over the result sequence (using ThrowingEnumerable)
- Proving that the second sequence isn't used until we've exhausted the first

No new collections or comparers needed this time though - it's all pretty straightforward. I haven't written any tests for null elements this time - I'm convinced enough by what I saw when implementing Distinct to believe they won't be a problem.

Let's implement it!

First things first: we can absolutely implement the simpler overload in terms of the more complex one, and I'll do the same for Except and Intercept. Here's the Union method:

```
public static IEnumerable<TSource> Union<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second)
{
    return Union(first, second, EqualityComparer<TSource>.Default);
}
```

So how do we implement the more complex overload? Well, I've basically been a bit disappointed by Union in terms of its conceptual weight. It doesn't really give us anything that the obvious combination of Concat and Distinct doesn't - so let's implement it that way first:

```
public static IEnumerable<TSource> Union<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second,
    IEqualityComparer<TSource> comparer)
{
    return first.Concat(second).Distinct(comparer);
}
```

The argument validation can be implemented by Concat with no problems here - the parameters have the same name, so any exceptions thrown will be fine in every way.

That's how I *think* about Union, but as I've mentioned before, I'd rather not actually see Concat and Distinct showing up in the stack trace - so here's the fuller implementation.

```
public static IEnumerable<TSource> Union<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second,
    IEqualityComparer<TSource> comparer)
{
    if (first == null)
    {
        throw new ArgumentNullException("first");
    }
    if (second == null)
    {
        throw new ArgumentNullException("second");
    }
    return UnionImpl(first, second, comparer ??
EqualityComparer<TSource>.Default);
}

private static IEnumerable<TSource> UnionImpl<TSource>(
    IEnumerable<TSource> first,
    IEnumerable<TSource> second,
    IEqualityComparer<TSource> comparer)
{
    HashSet<TSource> seenElements = new HashSet<TSource>(comparer);
    foreach (TSource item in first)
    {
        if (seenElements.Add(item))
        {
            yield return item;
        }
```

```
    }
    foreach (TSource item in second)
    {
        if (seenElements.Add(item))
        {
            yield return item;
        }
    }
}
```

That feels like an absurd waste of code when we can achieve the same result so simply, admittedly. This is the first time my resolve against implementing one operator in terms of completely different ones has wavered. Just looking at it in black and white (so to speak), I'm close to going over the edge...

Conclusion

Union was a disappointingly bland operator in my view. (Maybe I should start awarding operators marks out of ten for being interesting, challenging etc.) It doesn't feel like it's really earned its place in LINQ, as calls to Concat/Distinct can replace it so easily. Admittedly as I've mentioned in several other places, a lot of operators can be implemented in terms of each other - but rarely *quite* so simply.

Still, I think Intersect and Except should be more interesting.

Back to the table of contents.

Part 16 - Intersect (and build fiddling)

Okay, this is more like it - after the dullness of Union, Intersect has a new pattern to offer... and one which we'll come across repeatedly.

First, however, I should explain some more changes I've made to the solution structure...

Building the test assembly

I've just had an irritating time sorting out something I thought I'd fixed this afternoon. Fed up of accidentally testing against the wrong implementation, my two project configurations ("Normal LINQ" and "Edulinq implementation") now target different libraries from the test project: only the "Normal LINQ" configuration refers to System.Core, and only the "Edulinq implementation" configuration refers to the Edulinq project itself. Or so I thought. Unfortunately, msbuild automatically adds System.Core in unless you're careful. I had to add this into the "Edulinq implementation" property group part of my project file to avoid accidentally pulling in System.Core:

```
<AddAdditionalExplicitAssemblyReferences>false</AddAdditionalExplicitAssemblyReferences>
```

Unfortunately, at that point all the extension methods I'd written within the tests project - and the references to HashSet<T> - failed. I should have noticed them *not* failing before, and been suspicious. Hey ho.

Now I'm aware that you can add your own version of ExtensionAttribute, but I believe that can become a problem if at execution time you *do* actually load System.Core... which I will end up doing, as the Edulinq assembly itself references it (for HashSet<T> aside from anything else).

There may well be multiple solutions to this problem, but I've come up with one which appears to work:

- Add an Edulinq.TestSupport project, and refer to that from both configurations of Edulinq.Tests
- Make Edulinq.TestSupport refer to System.Core so it can use whatever collections it likes, as well as extension methods
- Put all the non-test classes (those containing extension methods, and the weird and wonderful collections) into the Edulinq.TestSupport project.
- Add a DummyClass to Edulinq.TestSupport in the namespace System.Linq, so that using directives within Edulinq.Tests don't need to be conditionalized

All seems to be working now - and finally, if I try to refer to an extension method I

haven't implemented in Edulinq yet, it will fail to compile instead of silently using the System.Core one. Phew. Now, back to Intersect...

What is it?

Intersect has a now-familiar pair of overloads:

```
public static IEnumerable<TSource> Intersect<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second)

public static IEnumerable<TSource> Intersect<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second,
    IEqualityComparer<TSource> comparer)
```

Fairly obviously, it computes the *intersection* of two sequences: the elements that occur in both "first" and "second". Points to note:

- Again, the first overload uses the default equality comparer for TSource, and the second overload does if you pass in "null" as the comparer, too.
- Neither "first" nor "second" can be null
- The method *does* use deferred execution - but in a way which may seem unusual at first sight
- The result sequence only contains distinct elements - even if an element occurs multiple times in both input sequences, it will only occur once in the result

Now for the interesting bit - exactly when the two sequences are used. MSDN claims this:

> When the object returned by this method is enumerated, Intersect enumerates first, collecting all distinct elements of that sequence. It then enumerates second, marking those elements that occur in both sequences. Finally, the marked elements are yielded in the order in which they were collected.

This is demonstrably incorrect. Indeed, I have a test which would fail under LINQ to Objects if this were the case. What *actually* happens is this:

- Nothing happens until the first result element is requested. I know I've said so already, but it's worth repeating.
- As soon as the first element of the result is requested, the *whole* of the "second" input sequence is read, as well as the first (and possibly more) elements of the "first" input sequence - enough to return the first result, basically.
- Results are read from the "first" input sequence *as and when they are required*. Only elements which were originally in the "second" input sequence and haven't already been yielded are returned.

We'll see this pattern of "greedily read the second sequence, but stream the first sequence" again when we look at Join... but let's get back to the tests.

What are we going to test?

Obviously I've got simple tests including:

- Argument validation
- Elements which occur multiple times in each sequence
- The overload without a comparer specified
- Specifying a null comparer
- Specifying a "custom" comparer (I'm using the case-insensitive string comparer again; news at 11)

However, the interesting tests are the ones which show how the sequences are actually consumed. Here they are:

```
[Test]
public void NoSequencesUsedBeforeIteration()
{
    var first = new ThrowingEnumerable();
    var second = new ThrowingEnumerable();
    // No exceptions!
    var query = first.Union(second);
    // Still no exceptions... we're not calling MoveNext.
    using (var iterator = query.GetEnumerator())
    {
    }
}

[Test]
public void SecondSequenceReadFullyOnFirstResultIteration()
{
    int[] first = { 1 };
    var secondQuery = new[] { 10, 2, 0 }.Select(x => 10 / x);

    var query = first.Intersect(secondQuery);
    using (var iterator = query.GetEnumerator())
    {
        Assert.Throws<DivideByZeroException>(() => iterator.MoveNext());
    }
}

[Test]
public void FirstSequenceOnlyReadAsResultsAreRead()
{
    var firstQuery = new[] { 10, 2, 0, 2 }.Select(x => 10 / x);
    int[] second = { 1 };

    var query = firstQuery.Intersect(second);
    using (var iterator = query.GetEnumerator())
    {
        // We can get the first value with no problems
        Assert.IsTrue(iterator.MoveNext());
```

```
        Assert.AreEqual(1, iterator.Current);

        // Getting at the *second* value of the result sequence requires
        // reading from the first input sequence until the "bad" division
        Assert.Throws<DivideByZeroException>(() => iterator.MoveNext());
    }
}
```

The first test just proves that execution really is deferred. That's straightforward.

The second test proves that the "second" input sequence is completely read as soon as we ask for our first result. If the operator had *really* read "first" and then started reading "second", it could have yielded the first result (1) without throwing an exception... but it didn't.

The third test proves that the "first" input sequence *isn't* read in its entirety before we start returning results. We manage to read the first result with no problems - it's only when we ask for the second result that we get an exception.

Let's implement it!

I have chosen to implement the same behaviour as LINQ to Objects rather than the behaviour described by MSDN, because it makes more sense to me. In general, the "first" sequence is given more importance than the "second" sequence in LINQ: it's generally the one which is streamed, with the second sequence being buffered if necessary.

Let's start off with the comparer-free overload - as before, it will just call the other one:

```
public static IEnumerable<TSource> Intersect<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second)
{
    return Intersect(first, second, EqualityComparer<TSource>.Default);
}
```

Now for the more interesting part. Obviously we'll have an argument-validating method, but what should we do in the guts? I find the duality between this and Distinct fascinating: there, we started with an empty set of elements, and tried to *add* each source element to it, yielding the element if we successfully added it (meaning it wasn't there before).

This time, we've effectively got a limited set of elements which we *can* yield, but we only want to yield each of them once - so as we see items, we can *remove* them from the set, yielding only if that operation was successful. The initial set is formed from the "second" input sequence, and then we just iterate over the "first" input sequence, removing and yielding appropriately:

```
public static IEnumerable<TSource> Intersect<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second,
    IEqualityComparer<TSource> comparer)
{
    if (first == null)
    {
        throw new ArgumentNullException("first");
    }
    if (second == null)
    {
        throw new ArgumentNullException("second");
    }
    return IntersectImpl(first, second, comparer ??
EqualityComparer<TSource>.Default);
}

private static IEnumerable<TSource> IntersectImpl<TSource>(
    IEnumerable<TSource> first,
    IEnumerable<TSource> second,
    IEqualityComparer<TSource> comparer)
{
    HashSet<TSource> potentialElements = new HashSet<TSource>(second,
comparer);
    foreach (TSource item in first)
    {
        if (potentialElements.Remove(item))
        {
            yield return item;
        }
    }
}
```

Next time we'll see a cross between the two techniques.

Ta-da - it all works as expected. I don't know whether this is how Intersect really works in LINQ to Objects, but I expect it does something remarkably similar.

Conclusion

After suffering from some bugs earlier today where my implementation didn't follow the documentation, it's nice to find some documentation which doesn't follow the real implementation :)

Seriously though, there's an efficiency point to be noted here. If you have two sequences, one long and one short, then it's much more efficient (mostly in terms of space) to use longSequence.Intersect(shortSequence) than shortSequence.Intersect(longSequence). The latter will require the whole of the long sequence to be in memory at once.

Next up - and I *might* just manage it tonight - our final set operator, Except.

Back to the table of contents.

Part 17 - Except

I'm really pleased with this one. Six minutes ago (at the time of writing this), I tweeted about the Intersect blog post. I then *started* writing the tests and implementation for Except... and I'm now done.

The tests are cut/paste/replace with a few tweaks - but it's the implementation that I'm most pleased with. You'll see what I mean later - it's beautiful.

What is it?

Except is our final set operator, with the customary two overloads:

```
public static IEnumerable<TSource> Except<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second)

public static IEnumerable<TSource> Except<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second,
    IEqualityComparer<TSource> comparer)
```

The result of calling Except is the elements of the first sequence which are *not* in the second sequence.

Just for completeness, here's a quick summary of the behaviour:

- The first overload uses the default equality comparer for TSource, and the second overload does if you pass in "null" as the comparer, too.
- Neither "first" nor "second" can be null
- The method *does* use deferred execution
- The result sequence only contains distinct elements; even if the first sequence contains duplicates, the result sequence won't

This time, the documentation doesn't say how "first" and "second" will be used, other than to describe the result as a set difference. In practice though, it's exactly the same as Intersect: when we ask for the first result, the "second" input sequence is fully evaluated, then the "first" input sequence is streamed.

What are we going to test?

I literally cut and paste the tests for Intersect, did a find/replace on Intersect/Except,

and then looked at the data in each test. In particular, I made sure that there were potential duplicates in the "first" input sequence which would have to be removed in the result sequence. I also tweaked the *details* of the data in the final tests shown before (which proved the way in which the two sequences were read) but the main thrust of the tests are the same.

Nothing to see here, move on...

Let's implement it!

I'm not even going to bother showing the comparer-free overload this time. It just calls the other overload, as you'd expect. Likewise the argument validation part of the implementation is tedious. Let's focus on the part which does the work. First, we'll think back to Distinct and Intersect:

- In Distinct, we started with an empty set and populated it as we went along, making sure we never returned anything already in the set
- In Intersect, we started with a populated set (from the second input sequence) and *removed* elements from it as we went along, *only* returning elements where an equal element was previously in the set

Except is simply a cross between the two: from Distinct we keep the idea of a "banned" set of elements that we add to; from Intersect we take the idea of starting off with a set populated from the second input element. Here's the implementation:

```
private static IEnumerable<TSource> ExceptImpl<TSource>(
    IEnumerable<TSource> first,
    IEnumerable<TSource> second,
    IEqualityComparer<TSource> comparer)
{
    HashSet<TSource> bannedElements = new HashSet<TSource>(second, comparer);
    foreach (TSource item in first)
    {
        if (bannedElements.Add(item))
        {
            yield return item;
        }
    }
}
```

The only differences between this and Intersect are:

- The name of the method (ExceptImpl instead of IntersectImpl)
- The name of the local variable holding the set (bannedElements instead of potentialElements)

- The method called in the loop (Add instead of Remove)

Isn't that just *wonderful*? Perhaps it shouldn't make me quite as happy as it does, but hey...

Conclusion

That concludes the set operators - and indeed my blogging for the night. It's unsurprising that all of the set operators have used a set implementation internally... but I've been quite surprised at just how simple the implementations all were. Again, the joy of LINQ resides in the ability for such simple operators to be combined in useful ways.

Back to the table of contents.

Part 18 - ToLookup

I've had a request to implement Join, which seems a perfectly reasonable operator to aim towards. However, it's going to be an awful lot easier to implement if we write ToLookup first. That will also help with GroupBy and GroupJoin, later on.

In the course of this post we'll create a certain amount of infrastructure - some of which we may want to tweak later on.

What is it?

ToLookup has four overloads, with these signatures:

```
public static ILookup<TKey, TSource> ToLookup<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector)

public static ILookup<TKey, TSource> ToLookup<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IEqualityComparer<TKey> comparer)

public static ILookup<TKey, TElement> ToLookup<TSource, TKey, TElement>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector)

public static ILookup<TKey, TElement> ToLookup<TSource, TKey, TElement>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    IEqualityComparer<TKey> comparer)
```

Essentially these boil down to two required parameters and two optional ones:

- The source is required, and must not be null
- The keySelector is required, and must not be null
- The elementSelector is optional, and defaults to an identity projection of a source element to itself. If it is specified, it must not be null.
- The comparer is optional, and defaults to the default equality comparer for the key type. It may be null, which is equivalent to specifying the default equality comparer for the key type.

Now we can just consider the most general case - the final overload. However, in

order to understand what ToLookup does, we have to know what ILookup<TKey, TElement> means - which in turn means knowing about IGrouping<TKey, TElement>:

```
public interface IGrouping<out TKey, out TElement> : IEnumerable<TElement>
{
    TKey Key { get; }
}

public interface ILookup<TKey, TElement> : IEnumerable<IGrouping<TKey,
TElement>>
{
    int Count { get; }
    IEnumerable<TElement> this[TKey key] { get; }
    bool Contains(TKey key);
}
```

The generics make these interfaces seem somewhat scary at first sight, but they're really not so bad. A *grouping* is simply a sequence with an associated key. This is a pretty simple concept to transfer to the real world - something like "Plays by Oscar Wilde" could be an IGrouping<string, Play> with a key of "Oscar Wilde". The key doesn't have to be "embedded" within the element type though - it would also be reasonable to have an IGrouping<string, string> representing just the *names* of plays by Oscar Wilde.

A *lookup* is essentially a map or dictionary where each key is associated with a sequence of values instead of a single one. Note that the interface is read-only, unlike IDictionary<TKey, TValue>. As well as looking up a single sequence of values associated with a key, you can also iterate over the whole lookup in terms of groupings (instead of the key/value pair from a dictionary). There's one other important difference between a lookup and a dictionary: if you ask a lookup for the sequence corresponding to a key which it doesn't know about, it will return an empty sequence, rather than throwing an exception. (A key which the lookup *does* know about will never yield an empty sequence.)

One slightly odd point to note is that while IGrouping is covariant in TKey and TElement, ILookup is invariant in both of its type parameters. While TKey has to be invariant, it would be reasonable for TElement to be covariant - to go back to our "plays" example, an IGrouping<string, Play> could be sensibly regarded as an IGrouping<string, IWrittenWork> (with the obvious type hierarchy). However, the interface declarations above are the ones in .NET 4, so that's what I've used in Edulinq.

Now that we understand the signature of ToLookup, let's talk about what it actually does. Firstly, it uses *immediate execution* - the lookup returned by the method is

effectively divorced from the original sequence; changes to the sequence after the method has returned won't change the lookup. (Obviously changes to the objects *within* the original sequence may still be seen, depending on what the element selector does, etc.) The rest is actually fairly straightforward, when you consider what parameters we have to play with:

- The keySelector is applied to each item in the input sequence. Keys are always compared for equality using the "comparer" parameter.
- The elementSelector is applied to each item in the input sequence, to project the item to the value which will be returned within the lookup.

To demonstrate this a little further, here are two applications of ToLookup - one using the first overload, and one using the last. In each case we're going to group plays by author and then display some information about them. Hopefully this will make some of the concepts I've described a little more concrete:

```
ILookup<string, Play> lookup = allPlays.ToLookup(play => play.Author);
foreach (IGrouping<string, Play> group in lookup)
{
    Console.WriteLine("Plays by {0}", group.Key);
    foreach (Play play in group)
    {
        Console.WriteLine("  {0} ({1})", play.Name, play.CopyrightDate);
    }
}

// Or...

ILookup<string, string> lookup = allPlays.ToLookup(play => play.Author,
                                                   play => play.Name,

StringComparer.OrdinalIgnoreCase);
foreach (IGrouping<string, string> group in lookup)
{
    Console.WriteLine("Plays by {0}:", group.Key);
    foreach (string playName in group)
    {
        Console.WriteLine("  {0}", playName);
    }
}

// And demonstrating looking up by a key:
IEnumerable<string> playsByWilde = lookup["Oscar Wilde"];
Console.WriteLine("Plays by Oscar Wilde: {0}",
                  string.Join("; ", playsByWilde));
```

Note that although I've used explicit typing for all the variables here, I would *usually* use implicit typing with var, just to keep the code more concise.

Now we just have the small matter of working out how to go from a key/element pair for each item, to a data structure which lets us look up sequences by key.

Before we move onto the implementation and tests, there are two aspects of ordering to consider:

- How are the groups within the lookup ordered?
- How are the elements within each group ordered?

The documentation for ToLookup is silent on these matters - but the docs for the closely-related GroupBy operator are more specific:

> The IGrouping<TKey, TElement> objects are yielded in an order based on the order of the elements in source that produced the first key of each IGrouping<TKey, TElement>. Elements in a grouping are yielded in the order they appear in source.

Admittedly "in an order based on..." isn't as clear as it might be, but I think it's reasonable to make sure that our implementation yields groups such that the first group returned has the same key as the first element in the source, and so on.

What are we going to test?

I've actually got relatively few tests this time. I test each of the overloads, but not terribly exhaustively. There are three tests worth looking at though. The first two show the "eager" nature of the operator, and the final one demonstrates the most complex overload by grouping people into families.

```
[Test]
public void SourceSequenceIsReadEagerly()
{
    var source = new ThrowingEnumerable();
    Assert.Throws<InvalidOperationException>(() => source.ToLookup(x => x));
}

[Test]
public void ChangesToSourceSequenceAfterToLookupAreNotNoticed()
{
    List<string> source = new List<string> { "abc" };
    var lookup = source.ToLookup(x => x.Length);
    Assert.AreEqual(1, lookup.Count);

    // Potential new key is ignored
    source.Add("x");
    Assert.AreEqual(1, lookup.Count);

    // Potential new value for existing key is ignored
```

```
    source.Add("xyz");
    lookup[3].AssertSequenceEqual("abc");
}


[Test]
public void LookupWithCompararеAndElementSelector()
{
    var people = new[] {
        new { First = "Jon", Last = "Skeet" },
        new { First = "Tom", Last = "SKEET" }, // Note upper-cased name
        new { First = "Juni", Last = "Cortez" },
        new { First = "Holly", Last = "Skeet" },
        new { First = "Abbey", Last = "Bartlet" },
        new { First = "Carmen", Last = "Cortez" },
        new { First = "Jed", Last = "Bartlet" }
    };

    var lookup = people.ToLookup(p => p.Last, p => p.First,
StringComparer.OrdinalIgnoreCase);

    lookup["Skeet"].AssertSequenceEqual("Jon", "Tom", "Holly");
    lookup["Cortez"].AssertSequenceEqual("Juni", "Carmen");
    // The key comparer is used for lookups too
    lookup["BARTLET"].AssertSequenceEqual("Abbey", "Jed");

    lookup.Select(x => x.Key).AssertSequenceEqual("Skeet", "Cortez",
"Bartlet");
}
```

I'm sure there are plenty of other tests I could have come up with. If you want to see any ones in particular (either as an edit to this post if they already exist in source control, or as entirely new tests), please leave a comment.

EDIT: It turned out there were a few important tests missing... see the addendum.

Let's implement it!

There's quite a lot of code involved in implementing this - but it should be reusable later on, potentially with a few tweaks. Let's get the first three overloads out of the way to start with:

```
public static ILookup<TKey, TSource> ToLookup<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector)
{
    return source.ToLookup(keySelector, element => element,
EqualityComparer<TKey>.Default);
}
```

```
public static ILookup<TKey, TSource> ToLookup<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IEqualityComparer<TKey> comparer)
{
    return source.ToLookup(keySelector, element => element, comparer);
}

public static ILookup<TKey, TElement> ToLookup<TSource, TKey, TElement>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector)
{
    return source.ToLookup(keySelector, elementSelector,
EqualityComparer<TKey>.Default);
}
```

Now we can just worry about the final one. Before we implement that though, we're definitely going to need an implementation of IGrouping. Let's come up with a really simple one to start with:

```
internal sealed class Grouping<TKey, TElement> : IGrouping<TKey, TElement>
{
    private readonly TKey key;
    private readonly IEnumerable<TElement> elements;

    internal Grouping(TKey key, IEnumerable<TElement> elements)
    {
        this.key = key;
        this.elements = elements;
    }

    public TKey Key { get { return key; } }

    public IEnumerator<TElement> GetEnumerator()
    {
        return elements.GetEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

How do we feel about this? Well, groupings should be immutable. We have no guarantee that the "values" sequence won't be changed after creation - but it's an internal class, so we've just got to be careful about how we use it. We also have to be careful that we don't use a mutable sequence type which allows a caller to get from

the iterator (the IEnumerator<TElement> returned by GetEnumerator) back to the sequence and then mutate it. In our case we're actually going to use List<T> to provide the sequences, and while List<T>.Enumerator is a public type, it doesn't expose the underlying list. Of course a caller could use reflection to mess with things, but we're not going to try to protect against that.

Okay, so now we can pair a sequence with a key... but we still need to implement ILookup. This is where there are multiple options. We want our lookup to be immutable, but there are various degrees of immutability we could use:

- Mutable internally, immutable in public API
- Mutable privately, immutable to the internal API
- Totally immutable, even within the class itself

The first option is the simplest to implement, and it's what I've gone for at the moment. I've created a Lookup class which is allows a key/element pair to be added to it from within the Edulinq assembly. It uses a Dictionary<TKey, List<TElement>> to map the keys to sequences efficiently, and a List<TKey> to remember the order in which we first saw the keys. Here's the complete implementation:

```
internal sealed class Lookup<TKey, TElement> : ILookup<TKey, TElement>
{
    private readonly Dictionary<TKey, List<TElement>> map;
    private readonly List<TKey> keys;

    internal Lookup(IEqualityComparer<TKey> comparer)
    {
        map = new Dictionary<TKey, List<TElement>>(comparer);
        keys = new List<TKey>();
    }

    internal void Add(TKey key, TElement element)
    {
        List<TElement> list;
        if (!map.TryGetValue(key, out list))
        {
            list = new List<TElement>();
            map[key] = list;
            keys.Add(key);
        }
        list.Add(element);
    }

    public int Count
    {
        get { return map.Count; }
    }

    public IEnumerable<TElement> this[TKey key]
```

```
    {
        get
        {
            List<TElement> list;
            if (!map.TryGetValue(key, out list))
            {
                return Enumerable.Empty<TElement>();
            }
            return list.Select(x => x);
        }
    }

    public bool Contains(TKey key)
    {
        return map.ContainsKey(key);
    }

    public IEnumerator<IGrouping<TKey, TElement>> GetEnumerator()
    {
        return keys.Select(key => new Grouping<TKey, TElement>(key,
map[key]))
                   .GetEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

All of this is really quite straightforward. Note that we provide an equality comparer to the constructor, which is then passed onto the dictionary - that's the only thing that needs to know how to compare keys.

There are only two points of interest, really:

- In the indexer, we don't return the list itself - that would allow the caller to mutate it, after casting it to List<TElement>. Instead, we just call Select with an identity projection, as a simple way of inserting a sort of "buffering" layer between the list and the caller. There are other ways of doing this, of course - including implementing the indexer with an iterator block.
- In GetEnumerator, we're retaining the key order by using our list of keys and performing a lookup on each key.

We're currently creating the new Grouping objects lazily - which will lead to fewer of them being created if the caller doesn't actually iterate over the lookup, but more of them if the caller iterates over it several times. Again, there are alternatives here - but without any good information about where to make the trade-off, I've just gone for the simplest code which works for the moment.

One last thing to note about Lookup - I've left it internal. In .NET, it's actually public - but the only way of getting at an instance of it is to call ToLookup and then cast the result. I see no particular reason to make it public, so I haven't.

Now we're finally ready to implement the last ToLookup overload - and it becomes pretty much trivial:

```
public static ILookup<TKey, TElement> ToLookup<TSource, TKey, TElement>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    IEqualityComparer<TKey> comparer)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    if (keySelector == null)
    {
        throw new ArgumentNullException("keySelector");
    }
    if (elementSelector == null)
    {
        throw new ArgumentNullException("elementSelector");
    }

    Lookup<TKey, TElement> lookup = new Lookup<TKey, TElement>(comparer ??
EqualityComparer<TKey>.Default);

    foreach (TSource item in source)
    {
        TKey key = keySelector(item);
        TElement element = elementSelector(item);
        lookup.Add(key, element);
    }
    return lookup;
}
```

When you look past the argument validation, we're just creating a Lookup, populating it, and then returning it. Simple.

Thread safety

Something I haven't addressed anywhere so far is thead safety. In particular, although all of this is nicely immutable when viewed from a single thread, I have a nasty feeling that *theoretically*, if the return value of our implementation of ToLookup was exposed to another thread, it could potentially observe the internal mutations we're making

here, as we're not doing anything special in terms of the memory model here.

I'm basically scared by lock-free programming these days, unless I'm using building blocks provided by someone else. While investigating the exact guarantees offered here would be interesting, I don't think it would really help with our understanding of LINQ as a whole. I'm therefore declaring thread safety to be out of scope for Edulinq in general :)

Conclusion

So that's ToLookup. Two new interfaces, two new classes, all for one new operator... so far. We can reuse almost all of this in Join though, which will make it very simple to implement. Stay tuned...

Addendum

It turns out that I missed something quite important: ToLookup has to handle null keys, as do various other LINQ operators (GroupBy etc). We're currently using a Dictionary<TKey, TValue> to organize the groups... and that doesn't support null keys. Oops.

So, first steps: write some tests proving that it fails as we expect it to. Fetch by a null key of a lookup. Include a null key in the source of a lookup. Use GroupBy, GroupJoin and Join with a null key. Watch it go bang. That's the easy part...

Now, we *can* do all the special-casing in Lookup itself - but it gets ugly. Our Lookup code was pretty simple before; it seems a shame to spoil it with checks everywhere. What we really need is a dictionary which *does* support null keys. Step forward NullKeyFriendlyDictionary, a new internal class in Edulinq. Now you might *expect* this to implement IDictionary<TKey, TValue>, but it turns out that's a pain in the neck. We hardly use any of the members of Dictionary - TryGetValue, the indexer, ContainsKey, and the Count property. That's it! So those are the only members I've implemented.

The class contains a Dictionary<TKey, TValue> to delegate *most* requests to, and it just handles null keys itself. Here's a quick sample:

```
internal sealed class NullKeyFriendlyDictionary<TKey, TValue>
{
    private readonly Dictionary<TKey, TValue> map;
    private bool haveNullKey = false;
    private TValue valueForNullKey;

    internal NullKeyFriendlyDictionary(IEqualityComparer<TKey> comparer)
```

```
    {
        map = new Dictionary<TKey, TValue>(comparer);
    }

    internal bool TryGetValue(TKey key, out TValue value)
    {
        if (key == null)
        {
            // This will be default(TValue) if haveNullKey is false,
            // which is what we want.
            value = valueForNullKey;
            return haveNullKey;
        }
        return map.TryGetValue(key, out value);
    }

    // etc
}
```

There's *one* potential flaw here, that I can think of: if you provide an IEqualityComparer<TKey> which treats some non-null key as equal to a null key, we won't spot that. If your source then contains both those keys, we'll end up splitting them into two groups instead of keeping them together. I'm not too worried about that - and I suspect there are all kinds of ways that could cause problems elsewhere anyway.

With this in place, and Lookup adjusted to use NullKeyFriendlyDictionary instead of just Dictionary, all the tests pass. Hooray!

At the same time as implementing this, I've tidied up Grouping itself - it now implements IList<T> itself, in a way which is immutable to the outside world. The Lookup now contains groups directly, and can return them very easily. The code is generally tidier, and anything using a group can take advantage of the optimizations applied to IList<T> - particularly the Count() operator, which is often applied to groups.

Back to the table of contents.

Part 19 - Join

You might expect that as joining is such a fundamental concept in SQL, it would be a complex operation to both describe and implement in LINQ. Fortunately, nothing could be further from the truth...

What is it?

Join has a mere two overloads, with the familiar pattern of one taking a custom equality comparer and the other not:

```
public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(
    this IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, TInner, TResult> resultSelector)

public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(
    this IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, TInner, TResult> resultSelector,
    IEqualityComparer<TKey> comparer)
```

These are daunting signatures to start with - four type parameters, and up to six normal method parameters (including the first "this" parameter indicating an extension method). *Don't panic*. It's actually quite simple to understand each individual bit:

- We have two sequences (outer and inner). The two sequences can have different element types (TOuter and TInner).
- For each sequence, there's also a *key selector* - a projection from a single item to the key for that item. Note that although the key type can be different from the two sequence types, there's only *one* key type - both key selectors have to return the same kind of key (TKey).
- An optional equality comparer is used to compare keys.
- A delegate (resultSelector) is used to project a pair of items whose keys are equal (one from each sequence) to the result type (TResult)

The idea is that we look through the two input sequences for pairs which correspond to equal keys, and yield one output element for each pair. This is an *equijoin*

operation: we can only deal with *equal* keys, not pairs of keys which meet some arbitrary condition. It's also an *inner join* in database terms - we will only see an item from one sequence if there's a "matching" item from the other sequence. I'll talk about mimicking *left joins* when I implement GroupJoin.

The documentation gives us details of the order in which we need to return items:

> Join preserves the order of the elements of outer, and for each of these elements, the order of the matching elements of inner.

For the sake of clarity, it's probably worth including a naive implementation which at least gives the right results in the right order:

```
// Bad implementation, only provided for the purposes of discussion
public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(
    this IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, TInner, TResult> resultSelector,
    IEqualityComparer<TKey> comparer)
{
    // Argument validation omitted
    foreach (TOuter outerElement in outer)
    {
        TKey outerKey = outerKeySelector(outerElement);
        foreach (TInner innerElement in inner)
        {
            TKey innerKey = innerKeySelector(innerElement);
            if (comparer.Equals(outerKey, innerKey))
            {
                yield return resultSelector(outerElement, innerElement);
            }
        }
    }
}
```

Aside from the missing argument validation, there are two important problems with this:

- It iterates over the inner sequence multiple times. I always advise anyone implementing a LINQ-like operator to only iterate over any input sequence once. There are sequences which are impossible to iterate over multiple times, or which may give different results each time. That's bad news.
- It always has a complexity of O(N * M) for N items in the inner sequence and M items in the outer sequence. Eek. Admittedly that's always a possible complexity - two sequences which have the same key for all elements will

always have that complexity - but in a typical situation we can do considerably better.

The real Join operator uses the same behaviour as Except and Intersect when it comes to how the input sequences are consumed:

- Argument validation occurs eagerly - both sequences and all the "selector" delegates have to be non-null; the comparer argument can be null, leading to the default equality comparer for TKey being used.
- The overall operation uses deferred execution: it doesn't iterate over either input sequence until something starts iterating over the result sequence.
- When MoveNext is called on the result sequence for the first time, it immediately consumes the *whole* of the inner sequence, buffering it.
- The outer sequence is streamed - it's only read one element at a time. By the time the result sequence has started yielding results from the second element of outer, it's forgotten about the first element.

We've started veering towards an implementation already, so let's think about tests.

What are we going to test?

I haven't bothered with argument validation tests this time - even with cut and paste, the 10 tests required to completely check everything feels like overkill.

However, I have tested:

- Joining two invalid sequences, but not using the results (i.e. testing deferred execution)
- The way that the two sequences are consumed
- Using a custom comparer
- Not specifying a comparer
- Using sequences of different types

See the source code for more details, but here's a flavour - the final test:

```
[Test]
public void DifferentSourceTypes()
{
    int[] outer = { 5, 3, 7 };
    string[] inner = { "bee", "giraffe", "tiger", "badger", "ox", "cat",
"dog" };

    var query = outer.Join(inner,
                           outerElement => outerElement,
                           innerElement => innerElement.Length,
```

```
                            (outerElement, innerElement) => outerElement + ":"
+ innerElement);
    query.AssertSequenceEqual("5:tiger", "3:bee", "3:cat", "3:dog",
"7:giraffe");
}
```

To be honest, the tests aren't very exciting. The implementation is remarkably simple though.

Let's implement it!

Trivial decision 1: make the comparer-less overload call the one with the comparer.

Trivial decision 2: use the split-method technique to validate arguments eagerly but defer the rest of the operation

That leaves us with the actual implementation in an iterator block, which is all I'm going to provide the code for here. So, what are we going to do?

Well, we know that we're going to have to read in the whole of the "inner" sequence - but let's wait a minute before deciding how to store it. We're then going to iterate over the "outer" sequence. For each item in the outer sequence, we need to find the key and then work out all the "inner" sequence items which match that key. Now, the idea of finding all the items in a sequence which match a particular key should sound familiar to you - that's exactly what a lookup is for. If we build a lookup from the inner sequence as our very first step, the rest becomes easy: we can fetch the sequence of matches, then iterate over them and yield the return value of calling result selector on the pair of elements.

All of this is easier to see in code than in words, so here's the method:

```
private static IEnumerable<TResult> JoinImpl<TOuter, TInner, TKey, TResult>(
    IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, TInner, TResult> resultSelector,
    IEqualityComparer<TKey> comparer)
{
    var lookup = inner.ToLookup(innerKeySelector, comparer);
    foreach (var outerElement in outer)
    {
        var key = outerKeySelector(outerElement);
        foreach (var innerElement in lookup[key])
        {
            yield return resultSelector(outerElement, innerElement);
        }
```

```
    }
}
```

Personally, I think this is rather beautiful... in particular, I like the way that it uses every parameter exactly once. Everything is just set up to work nicely.

But wait, there's more... If you look at the nested foreach loops, that should remind you of something: for each outer sequence element, we're computing a nested sequence, then applying a delegate to each pair, and yielding the result. That's almost exactly the definition of SelectMany! If only we had a "yield foreach" or "yield!" I'd be tempted to use an implementation like this:

```
private static IEnumerable<TResult> JoinImpl<TOuter, TInner, TKey, TResult>(
    IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, TInner, TResult> resultSelector,
    IEqualityComparer<TKey> comparer)
{
    var lookup = inner.ToLookup(innerKeySelector, comparer);
    // Warning: not really valid C#
    yield foreach outer.SelectMany(outerElement =>
lookup[outerKeySelector(outerElement)],
                                   resultSelector);
}
```

Unfortunately there's no such thing as a "yield foreach" statement. We can't just call SelectMany and return the result directly, because then we wouldn't be deferring execution. The best we can sensibly do is loop:

```
private static IEnumerable<TResult> JoinImpl<TOuter, TInner, TKey, TResult>(
    IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, TInner, TResult> resultSelector,
    IEqualityComparer<TKey> comparer)
{
    var lookup = inner.ToLookup(innerKeySelector, comparer);
    var results = outer.SelectMany(outerElement =>
lookup[outerKeySelector(outerElement)],
                                   resultSelector);
    foreach (var result in results)
    {
        yield return result;
    }
}
```

At that stage, I think I prefer the original form - which is still pretty simple, thanks to the lookup.

While I have avoided using other operators for implementations in the past, in this case it feels so completely natural, it would be silly *not* to use ToLookup to implement Join.

One final point before I wrap up for the night...

Query expressions

Most of the operators I've been implementing recently don't have any direct mapping in C# query expressions. Join does, however. The final test I showed before can also be written like this:

```
int[] outer = { 5, 3, 7 };
string[] inner = { "bee", "giraffe", "tiger", "badger", "ox", "cat", "dog" };

var query = from x in outer
            join y in inner on x equals y.Length
            select x + ":" + y;

query.AssertSequenceEqual("5:tiger", "3:bee", "3:cat", "3:dog", "7:giraffe");
```

The compiler will effectively generate the same code (although admittedly I've used shorter range variable names here - x and y instead of outerElement and innerElement respectively). In this case the resultSelector delegate it supplies is simply the final projection from the "select" clause - but if we had anything else (such as a where clause) between the join and the select, the compiler would introduce a *transparent identifier* to propagate the values of both x and y through the query. It looks like I haven't blogged explicitly about transparent identifiers, although I cover them in C# in Depth. Maybe once I've finished actually *implementing* the operators, I'll have a few more general posts on this sort of thing.

Anyway, the point is that for simple join clauses (as opposed to join...into) we've implemented everything we need to. Hoorah.

Conclusion

I spoiled some of the surprise around how easy Join would be to implement by mentioning it in the ToLookup post, but I'm still impressed by how neat it is. Again I should emphasize that this is due to the design of LINQ - it's got nothing to do with my

own prowess.

This won't be the end of our use of lookups though... the other grouping constructs can all use them too. I'll try to get them all out of the way before moving on to operators which feel a bit different...

Addendum

It turns out this wasn't quite as simple as I'd expected. Although ToLookup and GroupBy handle null keys without blinking, Join and GroupJoin *ignore* them. I had to write an alternative version of ToLookup which ignores null keys while populating the lookup, and then replace the calls of "ToLookup" in the code above with calls to "ToLookupNoNullKeys". This isn't documented anywhere, and is inconsistent with ToLookup/GroupBy. I've opened up a Connect issue about it, in the hope that it at least gets documented properly. (It's too late to change the behaviour now, I suspect.)

Back to the table of contents.

Part 20 - ToList

This morning I started writing the tests for GroupBy, prior to implementing it. That turned out to be a pain - really I wanted an easy way of getting at each element of the result (i.e. each result group). If only I had the ability to convert an arbitrary sequence into a query... I needed ToList. So, we enter a fairly brief diversion.

What is it?

ToList has a single, simple overload:

```
public static List<TSource> ToList<TSource>(this IEnumerable<TSource> source)
```

Fairly obviously, ToList converts the source sequence into a list. Some points to note:

- The signature specifies List<T>, not just IList<T>. Of course it could return a subclass of List<T>, but there seems little point.
- It uses *immediate execution* - nothing is deferred here
- The parameter (source) musn't be null
- It's optimized for the case when source implements ICollection<T>
- It *always* creates a new, independent list.

The last two points are worth a bit more discussion. Firstly, the optimization for ICollection<T> isn't documented, but it makes a lot of sense:

- List<T> stores its data in an array internally
- ICollection<T> exposes a Count property so the List<T> can create an array of exactly the right size to start with
- ICollection<T> exposes a CopyTo method so that the List<T> can copy all the elements into the newly created array in bulk

ToList always creates a new list for consistency. If it just returned the source parameter directly if it was already a List<T>, that would mean that changes to the source after calling ToList would *sometimes* be visible in the returned list and sometimes not... making it harder to reason about any code which used ToList.

What are we going to test?

I have tests for the bullet points listed above, and one extra test just to prove that it can work with lazily evaluated sequences as well as simple collections like arrays.

(The test uses a range followed by a projection.)

In order to test the optimization for ICollection<T>, I've implemented another collection like NonEnumerableList, but this time just NonEnumerableCollection. Again, this just delegates all ICollection<T> operations to a backing List<T>, but throws an exception if you try to call GetEnumerator(). The test then just calls ToList on a NonEnumerableCollection: as no exception is thrown, that proves that either the operation is optimized as we'd expect, or the exception is being swallowed. I think it's reasonable to assume that exceptions aren't swallowed in LINQ to Objects :)

Let's implement it!

This will probably be the simplest implementation in the whole of Edulinq:

```
public static List<TSource> ToList<TSource>(this IEnumerable<TSource> source)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    return new List<TSource>(source);
}
```

You may well be wondering what happened to the optimization... well, it's in the List<T> constructor. We just get it for free. Unfortunately *that's* not documented either... so we end up with an implementation which implements one undocumented optimization if List<T> implements another undocumented optimization :)

We can't actually do much better than that - we can't use ICollection<T>.CopyTo ourselves, as we don't have access to the underlying array of List<T>. We could perform *some* optimization by calling the List<T> constructor which specifies a capacity, and then call AddRange. That would at least prevent the list from having to resize itself, but it would still need to iterate over the whole collection instead of using the (potentially very fast) CopyTo method.

Conclusion

You may be wondering why we even need ToList, if we could just create a list by calling the constructor directly. The difference is that in order to call a constructor, you need to specify the element type as the type argument. When we use ToList, we can take advantage of type inference. In many cases this is just extremely convenient, but for anonymous types it's actually required. How can you end up with a strongly typed list of an anonymous type? It's easy with ToList, like this:

```
var query = Enumerable.Range(0, 10)
                      .Select(x => new { Value = x, Doubled = x * 2 });

var list = query.ToList();
```

Try doing that without an extension method or something similar. It's worth noting at this point that although there are similar methods for arrays and Dictionary, there's no equivalent for HashSet. It's incredibly easy to write, of course, and an obvious extension to LINQ to Objects - but it's not in the standard library. Maybe for .NET 5...

So, now that we've got ToList sorted, I can get back to GroupBy and its eight overloads - easy to implement, but hard to test simply and hard to describe clearly. Lucky me.

Back to the table of contents.

Part 21 - GroupBy

Okay, after the brief hiatus earlier, we're ready to group sequences.

What is it?

GroupBy has eight overloads, using up to four type parameters and up to five normal parameters. Those of a sensitive disposition may wish to turn away now:

```
public static IEnumerable<IGrouping<TKey, TSource>> GroupBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector)

public static IEnumerable<IGrouping<TKey, TSource>> GroupBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IEqualityComparer<TKey> comparer)

public static IEnumerable<IGrouping<TKey, TElement>> GroupBy<TSource, TKey,
TElement>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector)

public static IEnumerable<IGrouping<TKey, TElement>> GroupBy<TSource, TKey,
TElement>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    IEqualityComparer<TKey> comparer)

public static IEnumerable<TResult> GroupBy<TSource, TKey, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TKey, IEnumerable<TSource>, TResult> resultSelector)

public static IEnumerable<TResult> GroupBy<TSource, TKey, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TKey, IEnumerable<TSource>, TResult> resultSelector,
    IEqualityComparer<TKey> comparer)

public static IEnumerable<TResult> GroupBy<TSource, TKey, TElement, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    Func<TKey, IEnumerable<TElement>, TResult> resultSelector)
```

```
public static IEnumerable<TResult> GroupBy<TSource, TKey, TElement, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    Func<TKey, IEnumerable<TElement>, TResult> resultSelector,
    IEqualityComparer<TKey> comparer)
```

Okay, you can look back now. All the nastiness will go away soon, I promise. You see, this is yet another operator with overloads which effectively fill in defaults... although there's a twist here. Here are the various parameters and their meanings:

- source: the input sequence, as ever, of element type TSource.
- keySelector: a delegate to apply to each element in the input sequence to obtain a key (type TKey). The key is what decides which group the element is associated with.
- elementSelector (optional): a delegate to apply to each element (type TElement) to obtain the value which should be part of the relevant group. If it's not present, you can think of it as being the identity conversion.
- resultSelector (optional): a delegate to apply to each grouping to produce a final result (type TResult). See below for the difference between overloads which specify this and those which don't.
- comparer (optional): an equality comparer used to compare keys; groups are formed by equal keys. If unspecified or null, the default equality comparer for TKey is used.

Now, there's only one tricky bit here: resultSelector. In overloads where this is present, the result is type IEnumerable<TResult>; otherwise it's IEnumerable<IGrouping<TKey, TElement>>. That would make perfect sense *if* the type of resultSelector were Func<IGrouping<TKey, TElement>, TResult> - that would just make it effectively default to an identity conversion. However, the type is *actually* Func<TKey, IEnumerable<TElement>, TResult>. There's not a lot of difference between the two logically: basically you've got a key and a sequence of elements in both cases - but the disparity reduces the elegance of both the design and the implementation in my view. In fact, it's not *wholly* clear to me why the overloads with a resultSelector are required in the first place - you'll see why later.

Anyway, the basic operation should be reasonably familiar by now. We're going to look at each input value in turn, and map it to a key and a group element. We'll usually return a sequence of groups, where each group consists of the elements produced by input values with an equal key. If we've specified a resultSelector, that delegate will be presented with each key and all the members of the group associated with that key, and the final result sequence will consist of the return values of the delegate.

General behaviour:

- The operator is implemented with deferred execution, but it's not as deferred as some other operators. More below.
- All arguments other than comparer must not be null; this is validated eagerly.
- The source is read completely as soon as you start iterating over the results.

Basically this performs exactly the same steps as ToLookup, except:

- GroupBy uses deferred execution. Aside from anything else, this means that if you change the contents of "source" later and iterate over the results again, you'll see the changes (whereas the lookup returned by ToLookup is effectively independent of the source)
- The return value is always a sequence (whether it's a sequence of groups or the result of calling resultSelector). This means you can't perform arbitrary lookups on it later.

I've already quoted from the documentation when it comes to the ordering of the groups and the elements within the groups, but for completeness, here it is again:

> The IGrouping<TKey, TElement> objects are yielded in an order based on the order of the elements in source that produced the first key of each IGrouping<TKey, TElement>. Elements in a grouping are yielded in the order they appear in source.

When a resultSelector is present, the last sentence should be reinterpreted to consider the IEnumerable<TElement> sequence presented to resultSelector.

What are we going to test?

First let's talk about deferred execution. GroupBy *does* use deferred execution - but that it's not as deferred as usual. For most operators (such as Select) you can call GetEnumerator() on the result and it still won't actually start iterating over the input sequence until you call MoveNext() on the result's iterator. In GroupBy - at least in the "proper" implementation - it's the call to GetEnumerator() that does all the work.

In other words, my usual test like this wouldn't work:

```
// This won't work for GroupBy in LINQ to Objects
using (new ThrowingEnumerable().GroupBy(x => x).GetEnumerator())
{
    // No exception
}
```

In fact, I would probably argue that on reflection, my existing tests which check that

execution is deferred until the first call to MoveNext() are probably overzealous. For GroupBy, the tests are relatively liberal - they will work whether it's GetEnumerator() or MoveNext() which starts iterating over the input sequence. This is handy, as my implementation is currently slightly more deferred than that of LINQ to Objects :) Here are the relevant tests:

```
[Test]
public void ExecutionIsPartiallyDeferred()
{
    // No exception yet...
    new ThrowingEnumerable().GroupBy(x => x);
    // Note that for LINQ to Objects, calling GetEnumerator() starts
iterating
    // over the input sequence, so we're not testing that...
}

[Test]
public void SequenceIsReadFullyBeforeFirstResultReturned()
{
    int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
    // Final projection will throw
    var query = numbers.Select(x => 10 / x);

    var groups = query.GroupBy(x => x);
    Assert.Throws<DivideByZeroException>(() =>
    {
        using (var iterator = groups.GetEnumerator())
        {
            iterator.MoveNext();
        }
    });
}
```

I also have one test for each of the four overloads which doesn't specify a comparer. I haven't got any tests to ensure that the comparer is used properly - I felt they would be more effort than they were worth.

Here's the test for the most complicated overload, just to show how all the bits fit together:

```
[Test]
public void GroupByWithElementProjectionAndCollectionProjection()
{
    string[] source = { "abc", "hello", "def", "there", "four" };
    // This time "values" will be an IEnumerable<char>, the first character
of each
    // source string contributing to the group
    var groups = source.GroupBy(x => x.Length,
                                x => x[0],
```

```
                                    (key, values) => key + ":" + string.Join(";",
values));

    groups.AssertSequenceEqual("3:a;d", "5:h;t", "4:f");
}
```

Let's look at the parameters involved:

- source: just an array of strings
- keySelector: takes a string and maps it to its length. We will end up with three groups, as we have strings of length 3 ("abc" and "def"), 5 ("hello" and "there") and 4 ("four")
- elementSelector: takes a string and maps it to its first character.
- resultSelector: takes a key (the length) and a sequence of elements (the first characters of all the input strings of that length) and returns a string joining them all together

So the result for the first group is "3:a;d" - 3 is the key, 'a' and 'd' are the first letters of "abc" and "def", and they're joined together according to the resultSelector.

Finally, I also have a test demonstrating the use of GroupBy in a query expression, showing that these two queries are equivalent:

```
string[] source = { "abc", "hello", "def", "there", "four" };

var query1 = source.GroupBy(x => x.Length, x => x[0]);

var query2 = from x in source
             group x[0] by x.Length;
```

Okay, enough of the tests... what about the real code?

Let's implement it!

First, some interesting line counts:

- Lines in public method signatures: 36
- Lines implementing argument validation: 16
- Lines *just* delegating from one public overload to another: 6
- Lines of actual implementation: 7

Sad, isn't it? Still, there are things worth talking about.

Firstly, let's reduce the eight overloads to two: the two that both have elementSelector

and comparer specified. The six overloads which miss one or other of those parameters can simply be implemented by providing an identity conversion or the default key comparer. So that leaves us with this:

```
public static IEnumerable<IGrouping<TKey, TElement>> GroupBy<TSource, TKey,
TElement>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    IEqualityComparer<TKey> comparer)

public static IEnumerable<TResult> GroupBy<TSource, TKey, TElement, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    Func<TKey, IEnumerable<TElement>, TResult> resultSelector,
    IEqualityComparer<TKey> comparer)
```

Now *normally* I like to implement a method with fewer parameters by calling one with *more* parameters... but there's a slight problem in this case. We would have to provide a resultSelector which took a key and a sequence of elements, and creating a grouping from it. We *could* do that just by reusing our Grouping class... but there seems little reason to do that. In particular, suppose we implement it the other way round: given a grouping, we can easily extract the key (via the Key property) and the grouping itself is a sequence of the elements. In other words, we can implement the more complex method in terms of the simpler one:

```
public static IEnumerable<TResult> GroupBy<TSource, TKey, TElement, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    Func<TKey, IEnumerable<TElement>, TResult> resultSelector,
    IEqualityComparer<TKey> comparer)
{
    if (resultSelector == null)
    {
        throw new ArgumentNullException("resultSelector");
    }
    // Let the other GroupBy overload do the rest of the argument validation
    return source.GroupBy(keySelector, elementSelector, comparer)
                 .Select(group => resultSelector(group.Key, group));
}
```

The difference may seem somewhat arbitrary to start with, until we provide the final overload that we're delegating to. Surprise surprise, we can use ToLookup yet again:

```
public static IEnumerable<IGrouping<TKey, TElement>> GroupBy<TSource, TKey,
TElement>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    IEqualityComparer<TKey> comparer)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    if (keySelector == null)
    {
        throw new ArgumentNullException("keySelector");
    }
    if (elementSelector == null)
    {
        throw new ArgumentNullException("elementSelector");
    }
    return GroupByImpl(source, keySelector, elementSelector, comparer ??
EqualityComparer<TKey>.Default);
}

private static IEnumerable<IGrouping<TKey, TElement>> GroupByImpl<TSource,
TKey, TElement>(
    IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    IEqualityComparer<TKey> comparer)
{
    var lookup = source.ToLookup(keySelector, elementSelector, comparer);
    foreach (var result in lookup)
    {
        yield return result;
    }
}
```

You see, ILookup<TKey, TElement> already implements IEnumerable<IGrouping<TKey, TValue>> so we have no extra work to do. If we had implemented the two overloads the other way round by calling "new Grouping(key, sequence)" as a resultSelector, we'd have ended up with two "wrapper" layers of Grouping when we only need one. (Alternatively, we could have cast straight back to IGrouping, but that would have felt somewhat wrong, as well as requiring an execution-time check.)

Note that again, we could have used "yield foreach" to great effect if it had been available...

Now do you see why I believe this would have been more elegant if the type of resultSelector had been Func<IGrouping<TKey, TElement>, TResult> instead? That

way we could have just treated resultSelector as another optional parameter with a "default" value of the identity conversion... and all 7 simpler overloads could have delegated straight to the most complex one. Oh well.

Conclusion

Now we've done "Join" and "GroupBy", there's another very obvious operator to implement: GroupJoin. This (in conjunction with DefaultIfEmpty) is how the effect of left joins are usually achieved in LINQ. Let's see if ToLookup does what we need yet again...

Back to the table of contents.

Part 22 - GroupJoin

Another operator that was decidedly easy to implement - partly as I was able to just adapt everything from Join, including the tests. 15 minutes for tests + implementation... and no doubt considerably long writing it up.

What is it?

After the complexity of GroupBy, GroupJoin has a refreshingly small list of overloads:

```
public static IEnumerable<TResult> GroupJoin<TOuter, TInner, TKey, TResult>(
    this IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, IEnumerable<TInner>, TResult> resultSelector)

public static IEnumerable<TResult> GroupJoin<TOuter, TInner, TKey, TResult>(
    this IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, IEnumerable<TInner>, TResult> resultSelector,
    IEqualityComparer<TKey> comparer)
```

We can essentially ignore the first overload in the normal way - it just uses the default equality comparer for TKey. (You'll be pleased to hear that I think we're coming towards the end of the operators which use equality comparers, so I won't need to repeat that sort of sentence many more times.)

The best way to get to grips with what GroupJoin does is to think of Join. There, the overall idea was that we looked through the "outer" input sequence, found all the matching items from the "inner" sequence (based on a key projection on each sequence) and then yielded pairs of matching elements. GroupJoin is similar, except that instead of yielding pairs of elements, it yields a single result for each "outer" item based on that item and the sequence of matching "inner" items.

The fact that a result is yielded for *every* outer item is important: even if there are *no* matches, a result is still returned, just with an empty sequence. I'll come back to that later, thinking about "left joins".

In terms of deferred execution, input consumption etc, GroupJoin behaves very much like Join:

- Arguments are validated eagerly: everything other than comparer has to be non-null
- It uses deferred execution
- When you start iterating over the result, the "inner" sequence is immediately read all the way through
- The "outer" sequence is streamed: each time you read an item from the result sequence, one item from the outer sequence is read

What are we going to test?

To be absolutely honest, I cribbed the tests from Join, and just adapted the resultSelector and the expected results. It's simple enough to do: I was already using string concatenation to piece together the two items (outer and inner) for the Join tests; for GroupJoin I've just modified that to use string.Join (*not* the same as Enumerable.Join!) to build a semi-colon string delimited string of all the inner matches for that particular outer element.

Here's a sample test, in both the Join and GroupJoin versions:

```
[Test]
public void SimpleJoin()
{
    // We're going to join on the first character in the outer sequence item
    // being equal to the second character in the inner sequence item
    string[] outer = { "first", "second", "third" };
    string[] inner = { "essence", "offer", "eating", "psalm" };

    var query = outer.Join(inner,
                           outerElement => outerElement[0],
                           innerElement => innerElement[1],
                           (outerElement, innerElement) => outerElement + ":"
+ innerElement);

    // Note: no matches for "third"
    query.AssertSequenceEqual("first:offer", "second:essence",
"second:psalm");
}

[Test]
public void SimpleGroupJoin()
{
    // We're going to join on the first character in the outer sequence item
    // being equal to the second character in the inner sequence item
    string[] outer = { "first", "second", "third" };
    string[] inner = { "essence", "offer", "eating", "psalm" };

    var query = outer.GroupJoin(inner,
                           outerElement => outerElement[0],
```

```
                            innerElement => innerElement[1],
                            (outerElement, innerElements) => outerElement +
":" + string.Join(";", innerElements));

    query.AssertSequenceEqual("first:offer", "second:essence;psalm",
"third:");
}
```

Note how the match sequence for "first" contained only a single element and the match sequence for "second" contains two elements. There were no matches for "third", but we still get a result, just with an empty match sequence.

Let's implement it!

I was able to copy the implementation from Join as well, pretty much - I only had to simplify it to yield a single value instead of using a foreach loop.

Obviously the comparer-free overload calls into the one with the comparer, which then validates the arguments and calls into an iterator block method. If you've read all the rest of the posts in this series, you don't need to see that yet again - and if you haven't, just go back and look at some. So, what does the iterator block do? It uses a lookup again. It builds a lookup from the inner sequence, and then we just have to iterate over the outer sequence, applying resultSelector to each "outer element / matching inner element sequence" pair:

```
private static IEnumerable<TResult> GroupJoinImpl<TOuter, TInner, TKey,
TResult>(
    IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, IEnumerable<TInner>, TResult> resultSelector,
    IEqualityComparer<TKey> comparer)
{
    var lookup = inner.ToLookup(innerKeySelector, comparer);
    foreach (var outerElement in outer)
    {
        var key = outerKeySelector(outerElement);
        yield return resultSelector(outerElement, lookup[key]);
    }
}
```

Yup, Lookup makes things so simple it almost hurts... and that's without even trying hard. I'm sure we could pay some more attention to the design of Lookup and Grouping if we really wanted to. (I may revisit them later... they're clearly working for us at the moment.)

Query expressions and left joins

C# has direct support for GroupJoin when you use the "join ... into" syntax. Note that despite the use of "into" this is *not* a query continuation. Instead, it's just a way of giving a name to all the matches in the join. Here's an example, finding all the words of a given length:

```
int[] outer = { 5, 3, 7 };
string[] inner = { "bee", "giraffe", "tiger", "badger", "ox", "cat", "dog" };

var query = from x in outer
            join y in inner on x equals y.Length into matches
            select x + ":" + string.Join(";", matches);

query.AssertSequenceEqual("5:tiger", "3:bee;cat;dog", "7:giraffe");
```

Note that whereas in a simple "join" syntax, the "y" range variable would be available for the rest of the query, this time it's only "matches" that's available - basically "y" is only used to express the key selector (y.Length in this case).

Now think about left joins (or left outer joins) from SQL. From the linked Wikipedia page:

> The result of a *left outer join* (or simply **left join**) for table A and B always contains all records of the "left" table (A), even if the join-condition does not find any matching record in the "right" table (B). This means that if the ON clause matches 0 (zero) records in B, the join will still return a row in the resultâ€”but with NULL in each column from B.

So, how can we mimic this? Well, we could just use GroupJoin directly, and deal with the fact that if there are no matches, the match sequence will be empty. That's not quite like a left join though - it doesn't capture the idea of getting a null value for the unmatched result. We can use DefaultIfEmpty for that quite easily though. Here's an example adapted from the one above, where I introduce 4 into the outer sequence. There are no animals with 4 letters in the inner list, so for that item we end up with no values. I've used DefaultIfEmpty to make sure that we *do* get a value - and just to make it show up, I've specified a default value of the string literal "null" rather than just a null reference. Here's the test, including the results:

```
int[] outer = { 5, 3, 4, 7 };
string[] inner = { "bee", "giraffe", "tiger", "badger", "ox", "cat", "dog" };

var query = from x in outer
            join y in inner on x equals y.Length into matches
```

```
            select x + ":" + string.Join(";",
matches.DefaultIfEmpty("null"));

query.AssertSequenceEqual("5:tiger", "3:bee;cat;dog", "4:null", "7:giraffe");
```

Okay, that's getting closer... but we still need to deal with a whole sequence of results at a time. A normal left join still gives pairs of results... can we mimic *that*? Absolutely - just use GroupJoin and then SelectMany, represented in query expressions as a second "from" clause:

```
int[] outer = { 5, 3, 4, 7 };
string[] inner = { "bee", "giraffe", "tiger", "badger", "ox", "cat", "dog" };

var query = from x in outer
            join y in inner on x equals y.Length into matches
            from z in matches.DefaultIfEmpty("null")
            select x + ":" + z;
query.AssertSequenceEqual("5:tiger", "3:bee", "3:cat", "3:dog", "4:null",
"7:giraffe");
```

That's now genuinely close to what a SQL left join would do.

Conclusion

That's it for joins, I believe. I'm so glad I implemented ToLookup first - it made everything else trivial. Next up, I think we'll look at Take/TakeWhile/Skip/SkipWhile, which should be pleasantly simple.

Back to the table of contents.

Part 23 - Take/Skip/TakeWhile/SkipWhile

I genuinely expected these operators to be simple. At the time of this writing, I'm onto my third implementation of SkipWhile, struggling to find code which obviously expresses what's going on. I find it interesting how the simplest sounding operators sometimes end up being trickier to implement than one might think.

What are they?

Take, TakeWhile, Skip and SkipWhile form a natural group of operators. Together they are known as the *partitioning* operators. Here are the signatures:

```
public static IEnumerable<TSource> Take<TSource>(
    this IEnumerable<TSource> source,
    int count)

public static IEnumerable<TSource> TakeWhile<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)

public static IEnumerable<TSource> TakeWhile<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int, bool> predicate)


public static IEnumerable<TSource> Skip<TSource>(
    this IEnumerable<TSource> source,
    int count)

public static IEnumerable<TSource> SkipWhile<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)

public static IEnumerable<TSource> SkipWhile<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int, bool> predicate)
```

The behaviour is *reasonably* self-explanatory:

- source.Take(n) returns a sequence of the first n elements of source
- source.Skip(n) returns a sequence containing *all but* the first n elements of source
- source.TakeWhile(predicate) returns a sequence of the first elements of source

which match the given predicate; it stops as soon as the predicate fails to match
- source.SkipWhile(predicate) returns a sequence of *all but* the the first elements of source which match the given predicate; it starts yielding results as soon as the predicate fails to match

The only reason there are two overloads for TakeWhile and SkipWhile is so that you can provide a predicate which uses the index within the sequence, just like the overloads for Select and Where.

Each operator effectively partitions the input sequence into two parts, and yields either the first or second part. So for any "normal" collection (which returns the same results each time you iterate over it) the following are true:

- source.Take(n).Concat(source.Skip(n)) is equivalent to source
- source.TakeWhile(predicate).Concat(source.SkipWhile(predicate)) is equivalent to source (as noted in the comments, this is also assuming the predicate acts solely the same way each time it's given the same data, without any side-effects)

A few notes on general behaviour:

- The arguments are validated eagerly: source and predicate can't be null
- count *can* be negative (in which case it's equivalent to 0) and can be larger than the collection too. Basically any value is valid.
- Execution is deferred.
- The source sequence is streamed - there's no need to buffer anything.

What are we going to test?

One nice thing about the properties I described earlier is that it's very easy to convert tests for Take/TakeWhile into tests for Skip/SkipWhile: you just change the expected sequence in Skip to be everything in the source sequence which *wasn't* in the test for Take. For Take/Skip I've just used Enumerable.Range(0, 5) (i.e. 0, 1, 2, 3, 4) as the source sequence; for TakeWhile/SkipWhile I've used { "zero", "one", "two", "three", "four", "five" } - so each element is the word corresponding to the index. That makes it reasonably easy to write tests with a predicate involving the index.

I've tested:

- Argument validation
- Deferred execution
- For Skip/Take:

    - A negative count
    - A count of 0
    - A count to split the sequence into non-empty parts
    - A count exactly equal to the source length (5)
    - A count larger than the source length
- For SkipWhile/TakeWhile (both overloads in each case):
    - A predicate which fails on the first element
    - A predicate which matches some elements but not others
    - A predicate which matches all elements

The tests aren't generally interesting, but as it can be slightly tricky to get your head round TakeWhile and SkipWhile to start with, here's an example of each:

```
// In TakeWhileTest
[Test]
public void PredicateMatchingSomeElements()
{
    string[] source = { "zero", "one", "two", "three", "four", "five" };
    source.TakeWhile(x => x.Length < 5).AssertSequenceEqual("zero", "one",
"two");
}

// In SkipWhileTest
[Test]
public void PredicateMatchingSomeElements()
{
    string[] source = { "zero", "one", "two", "three", "four", "five" };
    source.SkipWhile(x => x.Length < 5).AssertSequenceEqual("three", "four",
"five");
}
```

There's been some discussion on Twitter about the best kind of test here, with suggestions of varying the data instead of the predicate, and using sequences of Boolean values. Personally I prefer the tests the way they are - I find this sequence of strings easy to work with, and the fact that it *is* a sequence of strings means you can't get the index/value parameter order wrong when providing a predicate which uses the indexes, unlike if you use an integer sequence. A Boolean sequence isn't clear enough for me - the result isn't obviously the first or second part of a sequence, whereas with these words, it's obvious what's going on... to me, at least. It's all a matter of personal preference though, and it's good to think about alternatives.

Let's implement them!

Let me present this in chronological order. I started off by implementing Take and Skip, as they sounded the simplest. Indeed, they're not too bad - here are the iterator

block parts, after argument validation in the public method, of course:

```
private static IEnumerable<TSource> TakeImpl<TSource>(
    this IEnumerable<TSource> source,
    int count)
{
    using (IEnumerator<TSource> iterator = source.GetEnumerator())
    {
        for (int i = 0; i < count && iterator.MoveNext(); i++)
        {
            yield return iterator.Current;
        }
    }
}

private static IEnumerable<TSource> SkipImpl<TSource>(
    this IEnumerable<TSource> source,
    int count)
{
    using (IEnumerator<TSource> iterator = source.GetEnumerator())
    {
        for (int i = 0; i < count; i++)
        {
            if (!iterator.MoveNext())
            {
                yield break;
            }
        }
        while (iterator.MoveNext())
        {
            yield return iterator.Current;
        }
    }
}
```

These really aren't too bad, although it's interesting to note that Skip is more complicated than Take: Skip has to deal with both sections of the sequence (the "ignore" part and the "yield" part) whereas Take can return as soon as it's finished with the "yield" part. We'll see the same pattern in TakeWhile/SkipWhile.

Speaking of which, let's look at them. There's a decision to make in terms of whether to implement the overload whose predicate *doesn't* use the index in terms of the one which does. It would certainly be simple, but I don't like the double level of indirection with one delegate calling another. It may well be irrational, given that I've been introducing identity delegates elsewhere - but it just didn't feel right. I've added it into the source code using conditional compilation just for completeness, but it's not the default implementation.

Now, the "Impl" bits of TakeWhile really weren't too bad:

```
private static IEnumerable<TSource> TakeWhileImpl<TSource>(
    IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    foreach (TSource item in source)
    {
        if (!predicate(item))
        {
            yield break;
        }
        yield return item;
    }
}

private static IEnumerable<TSource> TakeWhileImpl<TSource>(
    IEnumerable<TSource> source,
    Func<TSource, int, bool> predicate)
{
    int index = 0;
    foreach (TSource item in source)
    {
        if (!predicate(item, index))
        {
            yield break;
        }
        index++;
        yield return item;
    }
}
```

Again, we don't need to worry about the second part of the sequence - the part we're ignoring. We can just let the method complete, and the result iterator terminate. In particular, we don't care about the value that caused the predicate to return false. Now let's look at SkipWhile (again, just the interesting bits):

```
private static IEnumerable<TSource> SkipWhileImpl<TSource>(
    IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    using (IEnumerator<TSource> iterator = source.GetEnumerator())
    {
        while (iterator.MoveNext())
        {
            TSource item = iterator.Current;
            if (!predicate(item))
            {
                // Stop skipping now, and yield this item
                yield return item;
                break;
            }
```

```
        }
        while (iterator.MoveNext())
        {
            yield return iterator.Current;
        }
    }
}

private static IEnumerable<TSource> SkipWhileImpl<TSource>(
    IEnumerable<TSource> source,
    Func<TSource, int, bool> predicate)
{
    using (IEnumerator<TSource> iterator = source.GetEnumerator())
    {
        int index = 0;
        while (iterator.MoveNext())
        {
            TSource item = iterator.Current;
            if (!predicate(item, index))
            {
                // Stop skipping now, and yield this item
                yield return item;
                break;
            }
            index++;
        }
        while (iterator.MoveNext())
        {
            yield return iterator.Current;
        }
    }
}
```

The above code is my third attempt. You can look at the first two in the history of SkipWhile.cs - you may find one of those preferable. Other suggestions have been made, including ones using do/while... I must admit, the do/while option ends up being quite attractive, but again I have a general aversion to it as a construct. Part of the trickiness is that having found an element which makes the predicate return false, we have to yield that element before we move on to yielding the rest. It's easy enough to do, but the extra yield return statement feels ugly.

One point to note is that if the "skipping" part finishes due to a call to MoveNext returning false (rather than the predicate failing) then we'll call MoveNext again (at the start of the "yielding" loop). That's explicitly called out as being okay in the IEnumerator.MoveNext() documentation, so I'm happy enough to use it.

I'm certainly not happy about those implementations, but I'm confident they at least work. Now, let's revisit Skip and Take. Is there any way we can simplify them? Absolutely - all we need to do is use TakeWhile or SkipWhile with a predicate which

uses the index and compares it to the count we've been given:

```
public static IEnumerable<TSource> Take<TSource>(
    this IEnumerable<TSource> source,
    int count)
{
    // Buggy! See addendum...
    return source.TakeWhile((x, index) => index < count);
}

public static IEnumerable<TSource> Skip<TSource>(
    this IEnumerable<TSource> source,
    int count)
{
    return source.SkipWhile((x, index) => index < count);
}
```

TakeWhile and SkipWhile even do the appropriate argument validation for us. As you can see, I've become less averse to implementing one operator using another over time... both implementations are still in the source, using conditional compilation, but currently this short form is the one which is "active". I could be persuaded to change my mind :)

An optimized Skip?

Although most of these operations can't be sensibly optimized, it *would* make sense to optimize Skip when the source implements IList<T>. We can skip the skipping, so to speak, and go straight to the appropriate index. This wouldn't spot the case where the source was modified between iterations, which *may* be one reason it's not implemented in the framework as far as I'm aware. The optimization would look something like this:

```
var list = source as IList<TSource>;
if (list != null)
{
    // Make sure we don't use a negative index
    count = Math.Max(count, 0);
    // Note that "count" is the count of items to skip
    for (int index = count; index < list.Count; index++)
    {
        yield return list[index];
    }
    yield break;
}
```

Should Edulinq do this? Answers on a postcard...

EDIT: There's another optimization we can potentially make, even earlier. If we're given a count which is zero or negative, Skip is effectively pointless - so we can just return the original reference. This would come just after the argument validation, not in the iterator block:

```
if (count <= 0)
{
    return source;
}
```

Returning the original reference has some implications in terms of not isolating the return value from the original sequence, but it's worth considering. We could potentially do the same for Take, in the case where we know it's a list and the requested count is greater than or equal to the size of the list.

Using Skip and Take together

It's worth mentioning the most common use of Skip and Take, as well as an easy-to-make mistake. Skip and Take are ideal for paging. For example:

```
var pageItems = allItems.Skip(pageNumber * itemsPerPage)
                        .Take(itemsPerPage);
```

This simply skips as many items as it needs to, and only "takes" one page-worth of data afterwards. Very simple... but also easy to get wrong.

In many cases with LINQ, you can reorder operations with no harm - or with a compile-time error if you get them the wrong way round. Let's consider what happens if we get it wrong this time though:

```
// Bug! Bug! Bug!
var pageItems = allItems.Take(itemsPerPage)
                        .Skip(pageNumber * itemsPerPage);
```

This will work for the first page, but after that you'll *always* end up displaying an empty page: it will take the first page-worth of data, and then skip it all, leaving nothing to return.

Conclusion

That was more work than I expected it to be. Still, we've actually not got very many

LINQ operators still to cover. By my reckoning, we still have the following operators to implement:

- Sum, Average, Min, Max - I'm not looking forward to these, as they all have lots of overloads, and are likely to be tedious to test and implement.
- Cast and OfType
- ToArray and ToDictionary
- ElementAt and ElementAtOrDefault - I should possibly have covered these along with First/Single/Last etc
- SequenceEqual
- Zip (from .NET 4)
- Contains
- OrderBy/OrderByDescending/ThenBy/ThenByDescending (probably the most *interesting* remaining operators)
- Reverse

That's not too bad a list. I'll probably try to knock off ToArray tonight - it should be pretty simple.

Addendum

It turns out that my "simplification" for Take introduced a bug. If we only need to take two items, we should know that after we've taken the second one... we don't really need to ask the iterator to move to the third item, just to decide that we've finished. Doing so could have side effects - such as causing an exception to be thrown. That's exactly what we do when we use TakeWhile - we don't care what the first value after the stopping point is, but we *do* try to fetch it. Oops.

Fortunately the Mono tests picked this up for me, so I've added my own test for it, and gone back to the original implementation of Take which just used a counter. Both my own tests and Mono's now pass :)

Back to the table of contents.

Part 24 - ToArray

This really *is* a simple one. So simple I might even find the energy to implement ToDictionary as well tonight... we'll see. (EDIT: Oops. It became slightly less simple in the end, as I came up with the third implementation. Oh well.)

What is it?

ToArray is basically the equivalent of ToList, but producing an array instead of a list. It has a single signature:

```
public static TSource[] ToArray<TSource>(this IEnumerable<TSource> source)
```

Just to recap:

- It's another extension method, which is useful when we want to use type inference.
- It uses immediate execution - nothing is deferred, and the entire sequence is read before the method completes.
- It performs simple argument validation: source can't be null
- It creates an independent but shallow copy of the collection (so any changes to the objects referenced by source will be visible in the result, but not changes to source itself - and vice versa)
- It's optimized for ICollection<T>, although in a slightly different way to ToList.

What are we going to test?

Exactly the same as we did for ToList, basically - with one bit of care required. In our test for the source and result being independent of each other, we can't just create a new variable of type List<T> and call ToArray on it - because that would call the implementation in List<T> itself. I reckoned the easiest way of making this clear was to call the method directly as a normal static method, instead of using it as an extension method:

```
[Test]
public void ResultIsIndependentOfSource()
{
    List<string> source = new List<string> { "xyz", "abc" };
    // Make it obvious we're not calling List<T>.ToArray
    string[] result = Enumerable.ToArray(source);
    result.AssertSequenceEqual("xyz", "abc");
```

```
    // Change the source: result won't have changed
    source[0] = "xxx";
    Assert.AreEqual("xyz", result[0]);

    // And the reverse
    result[1] = "yyy";
    Assert.AreEqual("abc", source[1]);
}
```

One other interesting facet of the testing is that we can only *partially* test the optimization for ICollection<T>. We can make sure that it's optimized as far as not using the iterator (as per the previous test), but there's more optimization coming, and I haven't worked out how to test for that yet. You'll see what I mean when we come to implement it. Speaking of which...

Let's implement it!

We can make all our tests pass with a cheat: whatever the input type, convert it to a List<T> and then call ToArray on the result. Heck, if we call ToList to do the initial conversion, that will even do the argument validation for us and use the right parameter name in the exception:

```
// Implementation by Mr Cheaty MacCheaterson
public static TSource[] ToArray<TSource>(this IEnumerable<TSource> source)
{
    return source.ToList().ToArray();
}
```

Now I'm not averse to the general approach taken here - but there's actually a bit more optimization we can do.

Remember how ICollection<T> exposes Count and CopyTo, which the List<T> constructor uses when the input implements ICollection<T>? Well, that means that building a List<T> is relatively cheap for a collection - but calling ToArray on the list will still mean copying all the data out again (as List<T>.ToArray can't just return its own internal array - it has to create a copy). We can use exactly the same members ourselves, and avoid one level of copying:

```
// Somewhat better... though still not quite ideal
public static TSource[] ToArray<TSource>(this IEnumerable<TSource> source)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
```

```
    }
    ICollection<TSource> collection = source as ICollection<TSource>;
    if (collection != null)
    {
        TSource[] ret = new TSource[collection.Count];
        collection.CopyTo(ret, 0);
        return ret;
    }
    return new List<TSource>(source).ToArray();
}
```

That's pretty good now - except it still involves a copy from the List<T> into a new array every time. That's *almost always* appropriate, in fact... because unless the resulting list happened to expand its array to *exactly* the right size, we'd need to make a copy anyway. After all, we can't return an array that's too big. However, we *can* optimize for the "just the right size" case if we basically implement List<T>'s array expansion ourselves, leading to this:

```
public static TSource[] ToArray<TSource>(this IEnumerable<TSource> source)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }

    // Optimize for ICollection<T>
    ICollection<TSource> collection = source as ICollection<TSource>;
    if (collection != null)
    {
        TSource[] tmp = new TSource[collection.Count];
        collection.CopyTo(tmp, 0);
        return tmp;
    }

    // We'll have to loop through, creating and copying arrays as we go
    TSource[] ret = new TSource[16];
    int count = 0;
    foreach (TSource item in source)
    {
        // Need to expand...
        if (count == ret.Length)
        {
            Array.Resize(ref ret, ret.Length * 2);
        }
        ret[count++] = item;
    }

    // Now create another copy if we have to, in order to get an array of the
    // right size
    if (count != ret.Length)
    {
```

```
            Array.Resize(ref ret, count);
        }
        return ret;
    }
```

Is this level of optimization worth it? Probably not. I picked the starting size of 16 out of thin air (or dimly recalled initial counts for some collection or other - possibly Java's ArrayList<T>). Maybe we should triple the capacity rather than double it, just for laughs. It's all guesswork, really. The middle implementation feels like a more appropriate one to me, but with an obvious optimization just itching to be implemented, I thought I might as well provide the code. It's reasonably obviously correct, but it's just a bit longwinded for the marginal benefit over our second attempt.

It's even more annoying that I can't think of a way to test this easily - I could *benchmark* it of course, but that's not the same as unit testing it... I can't easily *prove* that we're optimizing either the ICollection<T> or "correct guess at size" cases.

Conclusion

It's always interesting to see what else springs to mind when I'm writing up the operator as opposed to just implementing it in Visual Studio. I'd got as far as the second implementation but not the third when I started this post.

It's possible that in the end, the 4-overload ToDictionary operator will actually end up being simpler than ToArray. Who'd have thought?

Back to the table of contents.

Part 25 - ToDictionary

This one ended up being *genuinely* easy to implement, although with lots of tests for different situations.

What is it?

ToDictionary has four overloads which look remarkably similar to the ones we used for ToLookup:

```
public static Dictionary<TKey, TSource> ToDictionary<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector)

public static Dictionary<TKey, TElement> ToDictionary<TSource, TKey,
TElement>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector)

public static Dictionary<TKey, TSource> ToDictionary<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IEqualityComparer<TKey> comparer)

public static Dictionary<TKey, TElement> ToDictionary<TSource, TKey,
TElement>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    IEqualityComparer<TKey> comparer)
```

Yes, it's the ever-popular combination of an optional key comparer and an optional element selector when mapping values.

ToDictionary does exactly what it says on the tin: it converts the source sequence to a dictionary. Each source item is mapped to a key, and then either the source item itself is used as the value or the element selector is applied to obtain one. A custom equality comparer can be used to create a dictionary which is (for example) case-insensitive when fetching keys.

The main difference between ToDictionary and ToLookup is that a dictionary can only store a single value per key, rather than a whole sequence of values. Of course, another difference is that Dictionary<TKey, TValue> is mutable concrete class,

whereas ILookup<TKey, TElement> is an interface usually implemented in an immutable fashion.

The normal sort of note:

- ToDictionary uses *immediate execution*: it reads the entire source sequence, and once it has returned, the dictionary is independent of the source
- source, keySelector and elementSelector mustn't be null
- comparer *can* be null, in which case the default equality comparer for the key type will be used
- No key can appear more than once (including non-identical but equal occurrences). This will cause an ArgumentException
- Null keys are prohibited, causing an ArgumentNullException. (This is slightly odd, as the ToDictionary method itself *doesn't* have a null argument at this point... really it should be a NullValueDerivedFromAnArgumentSomehowException, but that doesn't exist)
- Null values are allowed

Just a word on usage: one place I've seen ToDictionary used to reasonable effect is when the source is an IEnumerable<KeyValuePair<TKey, TValue>>, often from another dictionary. This lets you project all the values of a dictionary, or perhaps "trim" the dictionary in some ways... creating a new dictionary rather than modifying the existing one, of course. For example, here's a way of creating a new Dictionary<string, Person> from an existing one, but only keeping the entries which are adults:

```
var adults = nameToPersonMap.Where(pair => pair.Value.Age >= 18)
                            .ToDictionary(pair => pair.Key, pair =>
pair.Value);
```

Of course, this is just one use - it can be a pretty handy operator, to be frank.

What are we going to test?

Just the "null argument" validation tests gives us ten tests to start with, then:

- A test for each overload, always mapping a sequence of strings to a map using the first letter as the key; sometimes as a char and sometimes as a string (so we can easily use a case-insensitive comparer)
- A test for null keys (ArgumentNullException)
- A test for null values (no exception)
- A test for duplicate keys (ArgumentException)
- A test for a null comparer (uses the default)

Let's implement it!

In my implementation, the first three overloads all call the last - just using a default equality comparer or an identity element selector where necessary. Then after the argument validation, the implementation is simple:

```
public static Dictionary<TKey, TElement> ToDictionary<TSource, TKey,
TElement>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    IEqualityComparer<TKey> comparer)
{
    // Argument validation elided
    Dictionary<TKey, TElement> ret = new Dictionary<TKey, TElement>(comparer
?? EqualityComparer<TKey>.Default);
    foreach (TSource item in source)
    {
        ret.Add(keySelector(item), elementSelector(item));
    }
    return ret;
}
```

There's just one thing to be careful of - we have to use the Add method instead of the indexer to add entries to the dictionary. The Add method deals with duplicate and null keys in exactly the way we want, whereas the indexer would overwrite the existing entry if we had a duplicate key.

There's one thing my implementation *doesn't* do which it could if we really wanted - it doesn't optimize the case where we know the size beforehand, because the source implements ICollection<T>. We could use the Dictionary constructor which takes a capacity in that case, like this:

```
comparer = comparer ?? EqualityComparer<TKey>.Default;
ICollection<TSource> list = source as ICollection<TSource>;
var ret = list == null ? new Dictionary<TKey, TElement>(comparer)
                       : new Dictionary<TKey, TElement>(list.Count,
comparer);
```

After all, we know the eventual size of the dictionary will be *exactly* the count of the input sequence, in the success case.

Is this worth it? I'm not sure. It's not a huge change, admittedly... I'm quite tempted to put it in. Again, we'd have to benchmark it to see what difference it made (and I'd benchmark the .NET implementation too, of course) but a unit test won't show it up.

Thoughts welcome.

EDIT: I've now edited the code to include this optimization. I haven't benchmarked it though...

Conclusion

Gah - even when I thought I had this wrapped up before starting to write the post, that final bit of potential optimization *still* appeared out of nowhere.

Idle thought: if we all had to write up our code in blog posts as we went along, we might discover some interesting alternative designs and implementations.

That's it for tonight, and I may not have any time for posts tomorrow - we'll see.

Back to the table of contents.

Part 26a - IOrderedEnumerable

Implementing OrderBy/OrderByDescending/ThenBy/ThenByDescending isn't too bad, once you understand how the pattern works, but that's a slight conceptual leap. I'll take it one step at a time, first introducing some extra infrastructure (both public and internal), then implementing it simply and slowly, and finally optimizing it somewhat.

The sorting features of LINQ

As a query language, it's natural that LINQ allows you to sort sequences. Of course, collections have had the ability to sort their contents since v1.0 but LINQ is different in various ways:

- It expects you'll usually want to sort via projections, rather than on a "natural" comparison of the whole value. For example, should I come before or after my wife in a sorted list of people? It depends on whether you're ordering by first name, age, height, weight, or something else.
- It allows you to *easily* specify whether you want the ordering to be from the minimum value to the maximum (ascending) or from maximum to minimum (descending).
- It allows you to specify multiple ordering criteria - for example, order by last name and then first name.
- It doesn't sort in-place - instead, the sorting operators each return a new sequence which represents the results of the sort.
- It allows you to order *any* sequence implementing IEnumerable<T> - rather than Array.Sort being separate from List<T>.Sort, for example.
- The ordering operators in LINQ to Objects at least are *stable* - two equal items will be returned in the original order.

Other than stability (I'm at least unaware of any public, stable sort methods within the framework prior to .NET 3.5), all of this *can* be done using IComparer<T>, but it's a bit of a pain. Just as a reminder, IComparer<T> is an interface with a single method:

```
int Compare(T x, T y)
```

Unlike IEqualityComparer<T>, an IComparer<T> can *only* be asked to compare two items to determine which should come before the other in an ordered sequence. There's no notion of a hash code in IComparer<T>. All sorting in .NET is based on either IComparer<T> or Comparison<T>, which is the equivalent delegate. Like

IEqualityComparer<T>, a helper class (Comparer<T>) exists mostly to provide a default comparison based on the IComparable<T> and IComparable interfaces. (Unlike EqualityComparer<T>.Default, there's no "fall back" to a *universal* default comparison - if you try to compare two objects which have no comparison defined between them, an exception will be thrown.)

Composing comparers

Eventually, we'll need our ordering implementation to be built on top of IComparer<T>, so I'll introduce three helper classes to address the first three bullet points above. MiscUtil has more fully-developed versions of all of these classes - the code I've written for Edulinq is the bare minimum required for the project. Due to the way I'm testing (building the same tests against both Edulinq and the .NET framework version) I also don't have any tests for these classes - but they're simple enough that I'm pretty confident they'll work. They don't validate constructor arguments, as the public API is responsible for using them sensibly.

First, projection: we want to be able to specify a key selector and optionally a comparison between two keys - just as we already do for things like ToLookup and Join, but this time using IComparer<T> instead of IEqualityComparer<T>. Here's a helper class to build a comparison for two values of a "source" element type based on their keys:

```
internal class ProjectionComparer<TElement, TKey> : IComparer<TElement>
{
    private readonly Func<TElement, TKey> keySelector;
    private readonly IComparer<TKey> comparer;

    internal ProjectionComparer(Func<TElement, TKey> keySelector,
        IComparer<TKey> comparer)
    {
        this.keySelector = keySelector;
        this.comparer = comparer ?? Comparer<TKey>.Default;
    }

    public int Compare(TElement x, TElement y)
    {
        TKey keyX = keySelector(x);
        TKey keyY = keySelector(y);
        return comparer.Compare(keyX, keyY);
    }
}
```

As an example of how we might use this, we might want to order a List<Person> by last name in a case-insensitive way. For that, we might use:

```
IComparer<Person> comparer = new ProjectionComparer<Person, string>
    (p => p.LastName, StringComparer.CurrentCultureIgnoreCase);
```

Again, in public code I'd provide simpler ways of constructing the comparer to take advantage of type inference - but it isn't really worth it here.

As you can see, the comparer simply extracts the keys from both of the elements it's trying to compare, and delegates to the key comparer.

Next up is reversal: by reversing the order of the arguments into a comparer, we can effectively reverse the direction of a sort. Note that due to the stability of LINQ orderings, this is *not* quite the same as just reversing the final order of the elements: even when sorting by a key in a descending direction, elements with equal keys will be preserved in the original order. This actually drops out naturally when we let the comparer handle the direction - the ordering code can preserve stability without really knowing whether it's sorting in ascending or descending direction. Here's the reverse comparer:

```
internal class ReverseComparer<T> : IComparer<T>
{
    private readonly IComparer<T> forwardComparer;

    internal ReverseComparer(IComparer<T> forwardComparer)
    {
        this.forwardComparer = forwardComparer;
    }

    public int Compare(T x, T y)
    {
        return forwardComparer.Compare(y, x);
    }
}
```

It's incredibly simple, but there are two important points to note:

- It actually violates the contract of IComparer<T>, assuming that the forward comparer doesn't. The interface documentation states that null values should compare less than any non-null references, and obviously if we're reversing the original comparison, that won't be the case. I consider this to be a flaw in the documentation, to be honest: it makes sense to allow the comparer itself to decide how to handle nullity. The LINQ ordering documentation doesn't make any specific reference to this - it's unclear whether sorting by a key in a descending manner will leave elements with a null key at the start or the end. It turns out that the .NET implementation acts like my reverse comparer: it treats null like any other value, effectively.

- A naive implementation would call forwardComparer(x, y) and simply negate the return value. This would hide a subtle bug: if the forward comparer returned Int32.MinValue, *so would the reverse comparer*. In signed integer types, there are more negative numbers available than positive ones (e.g. the range of SByte is -128 to 127 inclusive) and negating the minimum value is effectively a no-op. By reversing the order of the arguments instead, we avoid the problem.

Our final helper class tackles the problem of creating a *compound* comparer, composed of a primary comparison and a secondary one. The secondary comparison is only used if the primary comparison considers two items to be equal - so for example, if we were comparing people by last name and then first name, the primary comparison would be the last name comparer, and the secondary comparison would be the first name comparer. Comparing "John Smith" and "Fred Bloggs" doesn't need to take the first names into account at all, but comparing "Dave Skeet" with "Jon Skeet" does.

If we want more than two comparers, we can simply treat one compound comparer as the primary comparsion for another compound comparer. For example, suppose we want to order by last name, then first name, then age. That's effectively "compound(compound(last name, first name), age)" in the obvious notation. Again, the code implementing a compound comparer is very simple, but will prove invaluable:

```
internal class CompoundComparer<T> : IComparer<T>
{
    private readonly IComparer<T> primary;
    private readonly IComparer<T> secondary;

    internal CompoundComparer(IComparer<T> primary,
        IComparer<T> secondary)
    {
        this.primary = primary;
        this.secondary = secondary;
    }

    public int Compare(T x, T y)
    {
        int primaryResult = primary.Compare(x, y);
        if (primaryResult != 0)
        {
            return primaryResult;
        }
        return secondary.Compare(x, y);
    }
}
```

(I *could* have used a conditional expression in the Compare method here, but this

time I felt it wasn't worth it.)

With these classes in place, we can move on to look at IOrderedEnumerable - and implement some of it.

Implementing IOrderedEnumerable<TElement>

When we eventually get to OrderBy, OrderByDescending, ThenBy and ThenByDescending (in the next post) they will all return IOrderedEnumerable<TElement>. By focusing on that first, we can actually make the LINQ operators themselves fairly trivial. The interface itself looks simple enough:

```
public interface IOrderedEnumerable<TElement> : IEnumerable<TElement>,
IEnumerable
{
    IOrderedEnumerable<TElement> CreateOrderedEnumerable<TKey>(
        Func<TElement, TKey> keySelector,
        IComparer<TKey> comparer,
        bool descending)
}
```

The important point to understand is that most developers will never need to call CreateOrderedEnumerable directly themselves - or even care about the interface. What most of us care about is that we can call ThenBy and ThenByDescending after we've called OrderBy or OrderByDescending - and that's what the interface enables. The ThenBy/ThenByDescending operators are extension methods on IOrderedEnumerable<TElement> instead of IEnumerable<T>, precisely because they have to be called on sequences which already have a primary ordering.

The CreateOrderedEnumerable method is precisely designed to create a new IOrderedEnumerable which represents the existing ordered sequence but with an extra secondary ordering applied.

The tricky bit to get your head round is that this sort of composition doesn't work the same way as our normal sequence chaining. Usually - for Where, Select etc - we fetch items from a "parent" sequence, perform some filtering or projection or whatever, and then yield the results. We can't do that to provide a secondary ordering without information about the primary ordering - we'd effectively have to know where the boundaries between the "primary sort keys" occur, so that we only reordered within one key. Even if we could do that, it would be relatively painful from a performance perspective.

Instead, we need to create an IOrderedEnumerable implementation which remembers the original, unordered sequence - and then builds up a compound

comparer as it goes. That way, we can then do all the ordering in one go, rather than each intermediate ordered sequence doing its own little bit.

This is possibly easier to show than to describe - my implementation of OrderedEnumerable is really fairly short. Here it is, just with the actual sorting code removed:

```
internal class OrderedEnumerable<TElement> : IOrderedEnumerable<TElement>
{
    private readonly IEnumerable<TElement> source;
    private readonly IComparer<TElement> currentComparer;

    internal OrderedEnumerable(IEnumerable<TElement> source,
        IComparer<TElement> comparer)
    {
        this.source = source;
        this.currentComparer = comparer;
    }

    public IOrderedEnumerable<TElement> CreateOrderedEnumerable<TKey>
        (Func<TElement, TKey> keySelector,
         IComparer<TKey> comparer,
         bool descending)
    {
        if (keySelector == null)
        {
            throw new ArgumentNullException("keySelector");
        }
        IComparer<TElement> secondaryComparer =
            new ProjectionComparer<TElement, TKey> (keySelector, comparer);
        if (descending)
        {
            secondaryComparer = new
ReverseComparer<TElement>(secondaryComparer);
        }
        return new OrderedEnumerable<TElement>(source,
            new CompoundComparer<TElement>(currentComparer,
secondaryComparer));
    }

    public IEnumerator<TElement> GetEnumerator()
    {
        // Perform the actual sorting here: basically load
        // "source" into a buffer, and then use a stable
        // sort algorithm with "currentComparer" to compare
        // any two elements
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
```

```
}
```

Note how when we create one IOrderedEnumerable from another, we don't store a reference to the original - it can be garbage-collected as far as we're concerned. We just care about the original (unordered) source, and the new compound comparer.

You can already see how useful our comparer helper classes are already. They allow us to focus on a single abstraction - IComparer<T> - and build interesting comparisons based on it, just as LINQ to Objects focuses on IEnumerable<T> and gives us interesting ways to work with sequences. Each aspect of the comparer we build is distinct:

- A projection comparer takes our key selector and key comparer, and builds a plain IComparer<TElement>
- If the secondary ordering is descending, we create a reverse comparer to wrap the projection comparer
- We then build a compound comparer from whatever comparison we were already using, and the newly created one

I suspect that this isn't the most efficient way to perform the comparison, to be honest: there's an awful lot of wrapping going on. However, I think it's probably the *clearest* way to represent things - and that's the goal of Edulinq.

Conclusion

We now have all the internal plumbing in place. There are two more tasks:

- Implementing the extension methods themselves
- Performing the sort

I'll cover those in the next two posts (26b and 26c). In some ways, they're likely to be less interesting than this post (unless I work out a particularly interesting sort algorithm to use). It's the *design* of sorting in LINQ which I find particularly neat - the way that it's tailored to what developers really need compared with the primitive Sort methods of old, and the elegant way in which those abilities are expressed. As ever, we've ended up with lots of small building blocks - but ones that can be combined to great effect.

Back to the table of contents.

Part 26b - OrderBy{,Descending}/ThenBy{,Descending}

Last time we looked at IOrderedEnumerable<TElement> and I gave an implementation we could use in order to implement the public extension methods within LINQ. I'm still going to do that in this post, but it's worth mentioning something else that's coming up in another part (26d) - I'm going to revisit my OrderedEnumerable implementation.

There may be trouble ahead...

A comment on the previous post mentioned how my comparer executes the keySelector on each element every time it makes a comparison. I didn't think of that as a particularly awful problem, until I thought of this sample query to rank people's favourite colours:

```
var query = people.GroupBy(p => p.FavouriteColour)
                  .OrderByDescending(g => g.Count())
                  .Select(g => g.Key);
```

Eek. Now every time we compare two elements, we have to count everything in a group. Ironically, I believe that counting the items in a group is fast using the LINQ to Objects implementation, but not in mine - something I may fix later on. But with LINQ to Objects, this wouldn't cause a problem in the first place!

There are ways to make this use an efficient key selector, of course - a simple Select before the OrderByDescending call would do fine... but it would be nicer if it wasn't a problem in the first place. Basically we want to extract the keys for each element *once*, and then compare them repeatedly when we need to. This would also allow us to shuffle a sequence using code such as this:

```
Random rng = new Random(); // Or get it from elsewhere...
var shuffled = collection.OrderBy(x => rng.NextDouble());
```

I'm not advocating that way of shuffling, admittedly - but it would be nice if it didn't cause significant problems, which it currently would, as the key selector is non-deterministic.

The interesting thing is that when I've finished today's post, I believe the code will obey all the *documented* behaviour of LINQ to Objects: there's nothing in the documentation about how often the key selector will be called. That doesn't mean it's a good idea to ignore this problem though, which is why I'll revisit OrderedEnumerable later. However, that's going to complicate the code somewhat... so while we're still getting to grips with how everything hangs together, I'm going to stick to my inefficient implementation.

Meanwhile, back to the actual LINQ operators for the day...

What are they?

OrderBy, OrderByDescending, ThenBy and ThenByDescending all have very similar overloads:

```
public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector)

public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IComparer<TKey> comparer)

public static IOrderedEnumerable<TSource> OrderByDescending<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector)

public static IOrderedEnumerable<TSource> OrderByDescending<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IComparer<TKey> comparer)

public static IOrderedEnumerable<TSource> ThenBy<TSource, TKey>(
    this IOrderedEnumerable<TSource> source,
    Func<TSource, TKey> keySelector)

public static IOrderedEnumerable<TSource> ThenBy<TSource, TKey>(
    this IOrderedEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IComparer<TKey> comparer)

public static IOrderedEnumerable<TSource> ThenByDescending<TSource, TKey>(
    this IOrderedEnumerable<TSource> source,
    Func<TSource, TKey> keySelector)

public static IOrderedEnumerable<TSource> ThenByDescending<TSource, TKey>(
    this IOrderedEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IComparer<TKey> comparer)
```

They're all extension methods, but ThenBy/ThenByDescending are extension methods on IOrderedEnumerable<T> instead of IEnumerable<T>.

We've already talked about what they do to some extent - each of them returns a sequence which is ordered according to the specified key. However, in terms of details:

- The source and keySelector parameters can't be null, and are validated eagerly.
- The comparer parameter (where provided) can be null, in which case the default comparer for the key type is used.
- They use deferred execution - the input sequence isn't read until it has to be.
- They read and buffer the entire input sequence when the result is iterated. Or

rather, they buffer the *original* input sequence - as I mentioned last time, when a compound ordered sequence (source.OrderBy(...).ThenBy(...).ThenBy(...)) is evaluated, the final query will go straight to the source used for OrderBy, rather than sorting separately for each key.

What are we going to test?

I have tests for the following:

- Deferred execution (using ThrowingEnumerable)
- Argument validation
- Ordering stability
- Simple comparisons
- Custom comparers
- Null comparers
- Ordering of null keys

In all of the tests which don't go bang, I'm using an anonymous type as the source, with integer "Value" and "Key" properties. I'm ordering using the key, and then selecting the value - like this:

```
[Test]
public void OrderingIsStable()
{
    var source = new[]
    {
        new { Value = 1, Key = 10 },
        new { Value = 2, Key = 11 },
        new { Value = 3, Key = 11 },
        new { Value = 4, Key = 10 },
    };
    var query = source.OrderBy(x => x.Key)
                      .Select(x => x.Value);
    query.AssertSequenceEqual(1, 4, 2, 3);
}
```

For ThenBy/ThenByDescending I have multiple key properties so I can test the interaction between the primary and secondary orderings. For custom key comparer tests, I have an AbsoluteValueComparer which simply compares the absolute values of the integers provided.

The "Value" property is always presented in ascending order (from 1) to make it easier to keep track of, and the "Key" properties are always significantly larger so we can't get confused between the two. I originally used strings for the keys in all tests, but then I found out that the default string comparer was culture-sensitive and didn't behave how I expected it to. (The default string *equality* comparer uses ordinal comparisons, which are rather less brittle...) I still use strings for the keys in nullity tests, but there I'm specifying the ordinal comparer.

I wouldn't claim the tests are exhaustive - by the time you've considered multiple orderings with possibly equal keys, different comparers etc the possibilities are overwhelming. I'm reasonably confident though (particularly after the tests found

some embarrassing bugs in the implementation). I don't think they're *hugely* readable either - but I was very keen to keep the value separated from the key, rather than just ordering by "x => x" in tests. If anyone fancies cloning the repository and writing better tests, I'd be happy to merge them :)

What I deliberately *don't* have yet is a test for how many times the key selector is executed: I'll add one before post 26d, so I can prove we're doing the right thing eventually.

Let's implement them!

We've got two bits of implementation to do before we can run the tests:

- The extension methods
- The GetEnumerator() method of OrderedEnumerable

The extension methods are extremely easy. All of the overloads without comparers simply delegate to the ones with comparers (using Comparer<TKey>.Default) and the remaining methods look like this:

```
public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IComparer<TKey> comparer)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    if (keySelector == null)
    {
        throw new ArgumentNullException("keySelector");
    }
    return new OrderedEnumerable<TSource>(source,
        new ProjectionComparer<TSource, TKey>(keySelector, comparer));
}

public static IOrderedEnumerable<TSource> OrderByDescending<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IComparer<TKey> comparer)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    if (keySelector == null)
    {
        throw new ArgumentNullException("keySelector");
    }
    IComparer<TSource> sourceComparer = new ProjectionComparer<TSource,
TKey>(keySelector, comparer);
    sourceComparer = new ReverseComparer<TSource>(sourceComparer);
    return new OrderedEnumerable<TSource>(source, sourceComparer);
}

public static IOrderedEnumerable<TSource> ThenBy<TSource, TKey>(
```

```
    this IOrderedEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IComparer<TKey> comparer)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    if (keySelector == null)
    {
        throw new ArgumentNullException("keySelector");
    }
    return source.CreateOrderedEnumerable(keySelector, comparer, false);
}
```

(To get ThenByDescending, just change the name of the method and change the last argument of CreateOrderedEnumerable to true.)

All very easy. I'm pretty sure I'm going to want to change the OrderedEnumerable constructor to accept the key selector and key comparer in the future (in 26d), which will make the above code even simpler. That can wait a bit though.

Now for the sorting part in OrderedEnumerable. Remember that we need a *stable* sort, so we can't just delegate to List<T>.Sort - at least, not without a bit of extra fiddling. (We could project to a type which contained the index, and add that onto the end of the comparer as a final tie-breaker.)

For the minute - and I swear it won't stay like this - here's the horribly inefficient (but easy to understand) implementation I've got:

```
public IEnumerator<TElement> GetEnumerator()
{
    // This is a truly sucky way of implementing it. It's the simplest I
could think of to start with.
    // We'll come back to it!
    List<TElement> elements = source.ToList();
    while (elements.Count > 0)
    {
        TElement minElement = elements[0];
        int minIndex = 0;
        for (int i = 1; i < elements.Count; i++)
        {
            if (currentComparer.Compare(elements[i], minElement) < 0)
            {
                minElement = elements[i];
                minIndex = i;
            }
        }
        elements.RemoveAt(minIndex);
        yield return minElement;
    }
}
```

We simply copy the input to a list (which is something we may well do in the final implementation - we certainly need to suck it all in somehow) and then repeatedly find the minimum element (favouring earlier elements over later ones, in order to

achieve stability), removing them as we go. It's an $O(n^2)$ approach, but hey - we're going for correctness first.

Conclusion

This morning, I was pretty confident this would be an easy and quick post to write. Since then, I've been found pain in the following items:

- Calling key selectors only once per element is more important than it might sound at first blush
- The default sort order for string isn't what I'd have guessed
- My (committed!) extension methods were broken, because I hadn't edited them properly after a cut and paste
- Writing tests for situations where there are lots of combinations is irritating

So far these have only extended my estimated number of posts for this group of operators to 4 (26a-26d) but who knows what the next few days will bring...

Back to the table of contents.

Part 26c - Optimizing OrderedEnumerable

Part 26b left us with a working implementation of the ordering operators, with two caveats:

- The sort algorithm used was awful
- We were performing the key selection on every comparison, instead of once to start with

Today's post is just going to fix the first bullet - although I'm pretty sure that fixing the second will require changing it again completely.

Choosing a sort algorithm

There are lots of sort algorithms available. In our case, we need the eventual algorithm to:

- Work on arbitrary pair-based comparisons
- Be stable
- Go like the clappers :)
- (Ideally) allow the first results to be yielded without performing *all* the sorting work, and without affecting the performance in cases where we *do* need all the results.

The final bullet it an interesting one to me: it's far from unheard of to want to get the "top 3" results from an ordered query. In LINQ to Objects we can't easily tell the Take operator about the OrderBy operator so that it could pass on the information, but we *can* potentially yield the first results before we've sorted everything. (In fact, we could add an extra interface specifically to enable this scenario, but it's not part of normal LINQ to Objects, and could introduce horrible performance effects with innocent-looking query changes.)

If we decide to implement sorting in terms of a naturally stable algorithm, that limits the choices significantly. I was rather interested in timsort, and may one day set about implementing it - but it looked far too complicated to introduce just for the sake of Edulinq.

The best bet seemed to be merge sort, which is reasonably easy to implement and has reasonable efficiency too. It requires extra memory and a fair amount of copying,

but we can probably cope with that.

We don't *have* to use a stable sort, of course. We could easily regard our "key" as the user-specified key plus the original index, and use that index as a final tie-breaker when comparing elements. That gives a stable *result* while allowing us to use any sorting algorithm we want. This may well be the approach I take eventually - especially as quicksort would allow us to start yielding results early in a fairly simple fashion. For the moment though, I'll stick with merge sort.

Preparing for merge sort

Just looking from the algorithm for merge sort, it's obvious that there will be a good deal of shuffling data around. As we want to make the implementation as fast as possible, that means it makes sense to use arrays to store the data. We don't need dynamic space allocation (after we've read all the data in, anyway) or any of the other features associated with higher-level collections. I'm aware that arrays are considered (somewhat) harmful, but purely for the internals of an algorithm which does so much data access, I believe they're the most appropriate solution.

We don't even need our arrays to be the right size - assuming we need to read in all the data before we start processing it (which will be true for this implementation of merge sort, but not for some other algorithms I may consider in the future) it's fine to use an oversized array as temporary storage - it's never going to be seen by the users, after all.

We've already got code which reads in all the data into a possibly-oversized array though - in the optimized ToArray code. So my first step was to extract out that functionality into a new *internal* extension method. This has to return a buffer containing all the data *and* give us an indication of the size. In .NET 4 I could use Tuple to return both pieces of data, but we can also just use an out parameter - I've gone for the latter approach at the moment. Here's the ToBuffer extension method:

```
internal static TSource[] ToBuffer<TSource>(this IEnumerable<TSource> source,
out int count)
{
    // Optimize for ICollection<T>
    ICollection<TSource> collection = source as ICollection<TSource>;
    if (collection != null)
    {
        count = collection.Count;
        TSource[] tmp = new TSource[count];
        collection.CopyTo(tmp, 0);
        return tmp;
    }
```

```
    // We'll have to loop through, creating and copying arrays as we go
    TSource[] ret = new TSource[16];
    int tmpCount = 0;
    foreach (TSource item in source)
    {
        // Need to expand...
        if (tmpCount == ret.Length)
        {
            Array.Resize(ref ret, ret.Length * 2);
        }
        ret[tmpCount++] = item;
    }
    count = tmpCount;
    return ret;
}
```

Note that I've used a local variable to keep track of the count in the loop near the end, only copying it into the output variable just before returning. This is due to a possibly-unfounded performance concern: we don't know where the variable will actually "live" in storage - and I'd rather not cause some arbitrary page of heap memory to be required all the way through the loop. This is a gross case of micro-optimization without evidence, and I'm tempted to remove it... but I thought I'd at least share my thinking.

This is only an internal API, so I'm trusting callers not to pass me a null "source" reference. It's possible that it would be a useful operator to expose at some point, but not just now. (If it were public, I would *definitely* use a local variable in the loop - otherwise callers could get weird effects by passing in a variable which could be changed elsewhere - such as due to side-effects within the loop. That's a totally avoidable problem, simply by using a local variable. For an internal API, I just need to make sure that I don't do anything so silly.)

Now ToArray needs to be changed to call ToBuffer, which is straightforward:

```
public static TSource[] ToArray<TSource>(this IEnumerable<TSource> source)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    int count;
    TSource[] ret = source.ToBuffer(out count);
    // Now create another copy if we have to, in order to get an array of the
    // right size
    if (count != ret.Length)
    {
        Array.Resize(ref ret, count);
    }
```

```
    return ret;
}
```

then we can prepare our OrderedEnumerable.GetEnumerator method for merging:

```
public IEnumerator<TElement> GetEnumerator()
{
    // First copy the elements into an array: don't bother with a list, as we
    // want to use arrays for all the swapping around.
    int count;
    TElement[] data = source.ToBuffer(out count);
    TElement[] tmp = new TElement[count];

    MergeSort(data, tmp, 0, count - 1);
    for (int i = 0; i < count; i++)
    {
        yield return data[i];
    }
}
```

The "tmp" array is for use when merging - while there *is* an in-place merge sort, it's more complex than the version where the "merge" step merges two sorted lists into a combined sorted list in temporary storage, then copies it back into the original list.

The arguments of 0 and count - 1 indicate that we want to sort the whole list - the parameters to my MergeSort method take the "left" and "right" boundaries of the sublist to sort - both of which are inclusive. Most of the time I'm more used to using exclusive upper bounds, but all the algorithm descriptions I found used inclusive upper bounds - so it made it easier to stick with that than try to "fix" the algorithm to use exclusive upper bounds everywhere. I think it highly unlikely that I'd get it all right without any off-by-one errors :)

Now all we've got to do is write an appropriate MergeSort method, and we're done.

Implementing MergeSort

I won't go through the *details* of how a merge sort works - read the wikipedia article for a pretty good description. In brief though, the MergeSort method guarantees that it will leave the specified portion of the input data sorted. It does this by splitting that section in half, and recursively merge sorting each half. It then merges the two halves by walking along two cursors (one from the start of each subsection) finding the smallest element out of the two at each point, copying that element into the temporary array and advancing just that cursor. When it's finished, the temporary storage will contain the sorted section, and it's copied back to the "main" array. The recursion has to stop at some point, of course - and in my implementation it stops if the section

has fewer than three elements.

Here's the MergeSort method itself first:

```
// Note: right is *inclusive*
private void MergeSort(TElement[] data, TElement[] tmp, int left, int right)
{
    if (right > left)
    {
        if (right == left + 1)
        {
            TElement leftElement = data[left];
            TElement rightElement = data[right];
            if (currentComparer.Compare(leftElement, rightElement) > 0)
            {
                data[left] = rightElement;
                data[right] = leftElement;
            }
        }
        else
        {
            int mid = left + (right - left) / 2;
            MergeSort(data, tmp, left, mid);
            MergeSort(data, tmp, mid + 1, right);
            Merge(data, tmp, left, mid + 1, right);
        }
    }
}
```

The test for "right > left" is part of a vanilla merge sort (if the section either has one element or none, we don't need to take any action), but I've optimized the common case of only two elements. All we need to do is swap the elements - and even then we only need to do so if they're currently in the wrong order. There's no point in setting up all the guff of the two cursors - or even have the slight overhead of a method call - for that situation.

Other than that one twist, this is a pretty standard merge sort. Now for the Merge method, which is slightly more complicated (although still *reasonably* straighforward):

```
private void Merge(TElement[] data, TElement[] tmp, int left, int mid, int
right)
{
    int leftCursor = left;
    int rightCursor = mid;
    int tmpCursor = left;
    TElement leftElement = data[leftCursor];
    TElement rightElement = data[rightCursor];
    // By never merging empty lists, we know we'll always have valid starting
points
```

```
    while (true)
    {
        // When equal, use the left element to achieve stability
        if (currentComparer.Compare(leftElement, rightElement) <= 0)
        {
            tmp[tmpCursor++] = leftElement;
            leftCursor++;
            if (leftCursor < mid)
            {
                leftElement = data[leftCursor];
            }
            else
            {
                // Only the right list is still active. Therefore tmpCursor
must equal rightCursor,
                // so there's no point in copying the right list to tmp and
back again. Just copy
                // the already-sorted bits back into data.
                Array.Copy(tmp, left, data, left, tmpCursor - left);
                return;
            }
        }
        else
        {
            tmp[tmpCursor++] = rightElement;
            rightCursor++;
            if (rightCursor <= right)
            {
                rightElement = data[rightCursor];
            }
            else
            {
                // Only the left list is still active. Therefore we can copy
the remainder of
                // the left list directly to the appropriate place in data,
and then copy the
                // appropriate portion of tmp back.
                Array.Copy(data, leftCursor, data, tmpCursor, mid -
leftCursor);
                Array.Copy(tmp, left, data, left, tmpCursor - left);
                return;
            }
        }
    }
}
```

Here, "mid" is the *exclusive* upper bound of the left subsection, and the *inclusive* lower bound of the right subsection... whereas "right" is the *inclusive* upper bound of the right subsection. Again, it's possible that this is worth tidying up at some point to be more consistent, but it's not too bad.

This time there's a little bit more special-casing. We take the approach that

whichever sequence runs out first (which we can detect as soon as the "currently advancing" cursor hits its boundary), we can optimize what still has to be copied. If the "left" sequence runs out first, then we know the remainder of the "right" sequence must already be in the correct place - so all we have to do is copy as far as we've written with tmpCursor back from the temporary array to the main array.

If the "right" sequence runs out first, then we can copy the rest of the "left" sequence directly into the right place (at the end of the section) and then again copy just what's needed from the temporary array back to the main array.

This is as fast as I've managed to get it so far (without delving into too many of the more complicated optimizations available) - and I'm reasonably pleased with it. I have no doubt that it could be improved significantly, but I didn't want to spend too much effort on it when I knew I'd be adapting everything for the key projection difficulty anyway.

Testing

I confess I don't know the best way to test sorting algorithms. I have two sets of tests here:

- A new project (MergeSortTest) where I actually implemented the sort before integrating it into OrderedEnumerable
- All my existing OrderBy (etc) tests

The new project also acts as a sort of benchmark - although it's pretty unscientific, and the key projection issue means the .NET implementation isn't *really* comparable with the Edulinq one at the moment. Still, it's a good indication of *very roughly* how well the implementation is doing. (It varies, interestingly enough... on my main laptop, it's about 80% slower than LINQ to Objects; on my netbook it's only about 5% slower. Odd, eh?) The new project sorts a range of sizes of input data, against a range of domain sizes (so with a small domain but a large size you're bound to get equal elements - this helps to verify stability). The values which get sorted are actually doubles, but we only sort based on the integer part - so if the input sequence is 1.3, 3.5, 6.3, 3.1 then we should get an output sequence of 1.3, 3.5, 3.1, 6.3 - the 3.5 and 3.1 are in that order due to stability, as they compare equal under the custom comparer. (I'm performing the "integer only" part using a custom comparer, but we could equally have used OrderBy(x => (int) x)).

Conclusion

One problem (temporarily) down, one to go. I'm afraid that the code in part 26d is likely to end up being pretty messy in terms of generics - and even then I'm likely to

talk about rather more options than I actually get round to coding.

Still, our simplistic model of OrderedEnumerable has served us well for the time being. Hopefully it's proved more useful educationally this way - I suspect that if I'd dived into the final code right from the start, we'd all end up with a big headache.

Back to the table of contents.

Part 26d - Fixing the key selectors, and yielding early

I feel I need a voice over. "Previously, on reimplementing LINQ to Objects..." Well, we'd got as far as a working implementation of OrderedEnumerable which didn't have terrible performance - *unless* you had an expensive key selector. Oh, and it didn't make use of the fact that we may only want the first few results.

Executing key selectors only once

Our first problem is to do with the key selectors. For various reasons (mentioned in part 26b) life is better if we execute the key selector once per input element. While we *can* do that with lazy evaluation, it makes more sense in my opinion to do it up-front. That means we need to separate out the key selector from the key comparer - in other words, we need to get rid of the handy ProjectionComparer we used to simplify the arguments to OrderBy/ThenBy/etc.

If we're going to keep the key selectors in a strongly typed way, that means our OrderedEnumerable (or at least *some* type involved in the whole business) needs to become generic in the key type. Let's bite the bullet and make it OrderedEnumerable. Now we have a slight problem right away in the fact that the "CreateOrderedEnumerable" method is generic, introducing a new type parameter TKey... so we shouldn't use TKey as the name of the new type parameter for OrderedEnumerable. We *could* rename the type parameter in the generic method implementation, but I'm becoming a big believer in leaving the signatures of methods alone when I implement an interface. For type parameters it's not too bad, but for normal parameters it can be awful if you mess around with the names - particularly for those using named arguments.

Thinking ahead, our single "key" type parameter in OrderedEnumerable could well end up being a composite key. After all, if we have OrderBy(...).ThenBy(...).ThenBy(...) we're going to have to have some way of representing the key formed by the three selectors. It makes sense to use a "nested" key type, where the key type of OrderedEnumerable is always the "composite key so far". Thus I named the type parameter TCompositeKey, and introduced an appropriate field. Here's the skeleton of the new class:

```
internal class OrderedEnumerable<TElement, TCompositeKey> :
IOrderedEnumerable<TElement>
{
```

```
    private readonly IEnumerable<TElement> source;
    private readonly Func<TElement, TCompositeKey> compositeSelector;
    private readonly IComparer<TCompositeKey> compositeComparer;

    internal OrderedEnumerable(IEnumerable<TElement> source,
        Func<TElement, TCompositeKey> compositeSelector,
        IComparer<TCompositeKey> compositeComparer)
    {
        this.source = source;
        this.compositeSelector = compositeSelector;
        this.compositeComparer = compositeComparer;
    }

    // Interface implementations here
}
```

(I'm aware this is very "stream of consciousness" - I'm assuming that presenting the decisions in the order in which I addressed them is a good way of explaining the necessary changes. Apologies if the style doesn't work for you.)

ThenBy and ThenByDescending don't have to change at all - they were already just using the interface. OrderBy and OrderByDescending become a little simpler, as we don't need to build the projection comparer. Here's the new version of OrderBy:

```
public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IComparer<TKey> comparer)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    if (keySelector == null)
    {
        throw new ArgumentNullException("keySelector");
    }
    return new OrderedEnumerable<TSource, TKey>
        (source, keySelector, comparer ?? Comparer<TKey>.Default);
}
```

Lovely - we just call a constructor, basically.

So far, so good. Now what about the implementation of IOrderedEnumerable? We should expect this to get messy, because there are three types of key involved:

- The current key type
- The secondary key type

- The composite key type

Currently we don't even have a type which can represent the composite key. We *could* use something like KeyValuePair<TKey, TValue>, but that doesn't really give the right impression. Instead, let's create our own simple type:

```
internal struct CompositeKey<TPrimary, TSecondary>
{
    private readonly TPrimary primary;
    private readonly TSecondary secondary;

    internal TPrimary Primary { get { return primary; } }
    internal TSecondary Secondary{ get { return secondary; } }

    internal CompositeKey(TPrimary primary, TSecondary secondary)
    {
        this.primary = primary;
        this.secondary = secondary;
    }
}
```

Now we can easily create a projection from two key selectors to a new one which selects a composite key. However, we'll need to do the same thing for a comparer. We *could* use the CompoundComparer class we created before, but that will end up with quite a bit of indirection. Instead, it would be nice to have a type to work directly with CompositeKey - something which *knew* it was dealing with comparers of different types, one for each part of the key.

We could create a completely separate top-level type for that... but specifying the type parameters again seems a bit daft when we can reuse them by simply creating a nested class within CompositeKey:

```
internal struct CompositeKey<TPrimary, TSecondary>
{
    // Other members as shown above

    internal sealed class Comparer : IComparer<CompositeKey<TPrimary,
TSecondary>>
    {
        private readonly IComparer<TPrimary> primaryComparer;
        private readonly IComparer<TSecondary> secondaryComparer;

        internal Comparer(IComparer<TPrimary> primaryComparer,
                          IComparer<TSecondary> secondaryComparer)
        {
            this.primaryComparer = primaryComparer;
            this.secondaryComparer = secondaryComparer;
        }
```

```
        public int Compare(CompositeKey<TPrimary, TSecondary> x,
                           CompositeKey<TPrimary, TSecondary> y)
        {
            int primaryResult = primaryComparer.Compare(x.Primary,
y.Primary);
            if (primaryResult != 0)
            {
                return primaryResult;
            }
            return secondaryComparer.Compare(x.Secondary, y.Secondary);
        }
    }
}
```

This may look a little odd to begin with, but the two types really are quite deeply connected.

Now that we can compose keys in terms of both selection and comparison, we can implement CreateOrderedEnumerable:

```
public IOrderedEnumerable<TElement> CreateOrderedEnumerable<TKey>(
    Func<TElement, TKey> keySelector,
    IComparer<TKey> comparer,
    bool descending)
{
    if (keySelector == null)
    {
        throw new ArgumentNullException("keySelector");
    }
    comparer = comparer ?? Comparer<TKey>.Default;
    if (descending)
    {
        comparer = new ReverseComparer<TKey>(comparer);
    }

    // Copy to a local variable so we don't need to capture "this"
    Func<TElement, TCompositeKey> primarySelector = compositeSelector;
    Func<TElement, CompositeKey<TCompositeKey, TKey>> newKeySelector =
        element => new CompositeKey<TCompositeKey,
TKey>(primarySelector(element), keySelector(element));

    IComparer<CompositeKey<TCompositeKey, TKey>> newKeyComparer =
        new CompositeKey<TCompositeKey, TKey>.Comparer(compositeComparer,
comparer);

    return new OrderedEnumerable<TElement, CompositeKey<TCompositeKey, TKey>>
        (source, newKeySelector, newKeyComparer);
}
```

I'm not going to pretend that the second half of the method is anything other than ghastly. I'm not sure I've ever written code which is so dense in type arguments. IComparer<CompositeKey<TCompositeKey, TKey>> is a particularly "fine" type. Ick.

However, it works - and once you've got your head round what each of the type parameters actually means at any one time, it's not really *complicated* code - it's just verbose and clunky.

The only bit which might require a bit of explanation is the primarySelector variable. I could certainly have just used compositeSelector within the lambda expression used to create the new key selector - it's not like it's going to change, after all. The memory benefits of not having a reference to "this" (where the intermediate OrderedEnumerable is likely to be eligible for GC collection immediately, in a typical OrderBy(...).ThenBy(...) call) are almost certainly not worth it. It just *feels right* to have both the primary and secondary key selectors in the same type, which is what will happen with the current code. They're both local variables, they'll be captured together, all will be well.

I hope you can see the parallel between the old code and the new code. Previously we composed a new (element-based) comparer based on the existing comparer, and a projection comparer from the method parameters. Now we're composing a new key selector and a new key comparer. It's all the same idea, just maintaining the split between key selection and key comparison.

Now let's sort...

So far, we haven't implemented GetEnumerator - and that's all. As soon as we've done that to our satisfaction, we're finished with ordering.

There are *several* approaches to how we could sort. Here are a few of them:

- Project each element to its key, and create a KeyValuePair for each item. Merge sort in the existing way to achieve stability. This will involve copying a lot of data around - particularly if the element and key types end up being large value types.
- Project each element to a { key, index } pair, and create another composite comparer which uses the index as a tie-breaker to achieve stability. This still involves copying keys around, but it means we could easily use a built-in sort (such as List<T>).
- Project each element to a key, and separately create an array of indexes (0, 1, 2, 3...). Sort the *indexes* by accessing the relevant key at any point, using indexes as tie-breakers. This requires a more fiddly sort, as we need to keep indexing into the indexes array.

- Build up "chunks" of sorted data as we read it in, keeping some number of chunks and merging them appropriate when we want to. We can then yield the results without ever performing a full sort, by effectively performing the "merge" operation of merge sort, just yielding values instead of copying them to temporary storage. (Obviously this is trivial with 2 chunks, but can be extended to more.)
- Do something involving a self-balancing binary tree :)

I decided to pick the middle option, using quicksort as the sorting algorithm. This comes with the normal problems of *possibly* picking bad pivots, but it's usually a reasonable choice. I believe there are cunning ways of improving the worst-case performance, but I haven't implemented any of those.

Here's the non-quicksort part of the code, just to set the scene.

```
public IEnumerator<TElement> GetEnumerator()
{
    // First copy the elements into an array: don't bother with a list, as we
    // want to use arrays for all the swapping around.
    int count;
    TElement[] data = source.ToBuffer(out count);

    int[] indexes = new int[count];
    for (int i = 0; i < indexes.Length; i++)
    {
        indexes[i] = i;
    }

    TCompositeKey[] keys = new TCompositeKey[count];
    for (int i = 0; i < keys.Length; i++)
    {
        keys[i] = compositeSelector(data[i]);
    }

    QuickSort(indexes, keys, 0, count - 1);

    for (int i = 0; i < indexes.Length; i++)
    {
        yield return data[indexes[i]];
    }
}
```

I could certainly have combined the first two loops - I just liked the separation provided in this code. One tiny micro-optimization point to note is that for each loop I'm using the Length property of the array rather than "count" as the upper bound, as I believe that will reduce the amount of array boundary checking the JIT will generate. I very much doubt that it's relevant, admittedly :) I've left the code here as it is in source

control - but looking at it now, I could certainly have used a foreach loop on the final yield part. We wouldn't be able to later, admittedly... but I'll come to that all in good time.

The actual quicksort part is reasonably standard except for the fact that I pass in both the arrays for both indexes and keys - usually there's just the one array which is being sorted. Here's the code for both the recursive call and the partition part:

```
private void QuickSort(int[] indexes, TCompositeKey[] keys, int left, int
right)
{
    if (right > left)
    {
        int pivot = left + (right - left) / 2;
        int pivotPosition = Partition(indexes, keys, left, right, pivot);
        QuickSort(indexes, keys, left, pivotPosition - 1);
        QuickSort(indexes, keys, pivotPosition + 1, right);
    }
}

private int Partition(int[] indexes, TCompositeKey[] keys, int left, int
right, int pivot)
{
    // Remember the current index (into the keys/elements arrays) of the
pivot location
    int pivotIndex = indexes[pivot];
    TCompositeKey pivotKey = keys[pivotIndex];

    // Swap the pivot value to the end
    indexes[pivot] = indexes[right];
    indexes[right] = pivotIndex;
    int storeIndex = left;
    for (int i = left; i < right; i++)
    {
        int candidateIndex = indexes[i];
        TCompositeKey candidateKey = keys[candidateIndex];
        int comparison = compositeComparer.Compare(candidateKey, pivotKey);
        if (comparison < 0 || (comparison == 0 && candidateIndex <
pivotIndex))
        {
            // Swap storeIndex with the current location
            indexes[i] = indexes[storeIndex];
            indexes[storeIndex] = candidateIndex;
            storeIndex++;
        }
    }
    // Move the pivot to its final place
    int tmp = indexes[storeIndex];
    indexes[storeIndex] = indexes[right];
    indexes[right] = tmp;
    return storeIndex;
}
```

It's interesting to observe how similar the quicksort and merge sort recursive parts are - both picking a midpoint, recursing on the left of it, recursing on the right of it, and performing some operation on the whole sublist. Of course the "some operation" is very different between partition and merge, and it occurs at a different time - but it's an interesting parallel nonetheless.

One significant difference between merge sort and quicksort is the use of the pivot. Once Partition has returned where the pivot element ended up, quicksort doesn't touch that element itself (we already know it will be in the right place). It recurses on the sublist entirely to the left of the pivot and the sublist entirely to the right of the pivot. Compare this with merge sort with recurses on two sublists which together comprise the whole list for that call.

The overloading of the word "index" here is unfortunate, but that is unfortunately life. Both sorts of "index" here really are indexes... you just need to keep an eye on which is which.

The final point to note is how we're using the indexes in the comparison, as a tie-break to keep stability. It's an ugly expression, but it does the job.

(As a small matter of language, I wasn't sure whether to use indexes or indices. I far prefer the former, so I used it. Having just checked in the dictionary, it appears both are correct. This reminds me of when I was writing C# in Depth - I could never decide between appendixes and appendices. Blech.)

Now, do you want to hear the biggest surprise I received last night? After I'd fixed up the compile-time errors to arrive at the code above, *it worked first time*. I'm not kidding. I'm not quite sure how I pulled that off (merge sort didn't take long either, but it did at least have a few tweaks to fix up) but it shocked the heck out of me. So, are we done? Well, not quite.

Yielding early

Just as a reminder, one of my aims was to be able to use iterator blocks to return some values to anyone iterating over the result stream without having to do *all* the sorting work. This means that in the case of calling OrderBy(...).Take(5) on a large collection, we can end up saving a lot of work... I hope!

This is currently fairly normal quicksort code, leaving the "dual arrays" aspect aside... but it's not quite amenable to early yielding. We're definitely *computing* the earliest results first, due to the order of the recursion - but we can't yield from the recursive method - iterator blocks just don't do that.

So, we'll have to fake the recursion. Fortunately, quicksort is only *directly* recursive - we don't need to worry about mutually recursive routines: A calling B which might call C or it might call back to A, etc. Instead, we can just keep a Stack<T> of "calls" to quicksort that we want to make, and execute the appropriate code within our GetEnumerator() method, so we can yield at the right point. Now in the original code, quicksort has four parameters, so you might expect our Stack<T> to have those four values within T too... but no! Two of those values are just the keys and indexes... and we already have those in two local variables. We only need to keep track of "right" and "left". Again, for the sake of clarity I decided to implement this using a custom struct - nested within OrderedEnumerable as there's no need for it to exist anywhere else:

```
private struct LeftRight
{
    internal int left, right;
    internal LeftRight(int left, int right)
    {
        this.left = left;
        this.right = right;
    }
}
```

Purists amongst you may curse at the use of internal fields rather than properties. I'm not bothered - this is a private class, and we're basically using this as a tuple. Heck, I would have used anonymous types if it weren't for two issues:

- I wanted to use Stack<T>, and there's no way of creating one of those for an anonymous type (without introducing more generic methods to use type inference)
- I wanted to use a struct - we'll end up creating a lot of these values, and there's simply no sense in them being individual objects on the heap. Anonymous types are always classes.

So, as a first step we can transform our code to use this "fake recursion" but still yield at the very end:

```
var stack = new Stack<LeftRight>();
stack.Push(new LeftRight(0, count - 1));
while (stack.Count > 0)
{
    LeftRight leftRight = stack.Pop();
    int left = leftRight.left;
    int right = leftRight.right;
    if (right > left)
    {
```

```
            int pivot = left + (right - left) / 2;
            int pivotPosition = Partition(indexes, keys, left, right, pivot);
            stack.Push(new LeftRight(pivotPosition + 1, right));
            stack.Push(new LeftRight(left, pivotPosition - 1));
        }
    }

    for (int i = 0; i < indexes.Length; i++)
    {
        yield return data[indexes[i]];
    }
```

We initially push a value of (0, count - 1) to simulate the call to QuickSort(0, count - 1) which started it all before. The code within the loop is very similar to the original QuickSort method, with three changes:

- We have to grab the next value of LeftRight from the stack, and then separate it into left and right values
- Instead of calls to QuickSort, we have calls to stack.Push
- We've reversed the order of the recursive calls: in order to sort the left sublist *first*, we have to push it onto the stack *last*.

Happy so far? We're getting very close now. All we need to do is work out when to yield. This is the bit which caused me the most headaches, until I worked out that the "if (right > left)" condition really meant "if we've got work to do"... and we're interested in the exact opposite scenario - when we *don't* have any work to do, as that means everything up to and including "right" is already sorted. There are two situations here: either right == left, i.e. we're sorting one element, or right == left - 1, which will occur if we picked a pivot which was the maximum or minimum value in the list at the previous recursive step.

It's taken me a little bit of thinking (and just running the code) to persuade me that we will *always* naturally reach a situation where we end up seeing right == count and right <= left, i.e. a place where we know we're completely done. But it's okay - it does happen.

It's not just a case of yielding the values between left and right though - because otherwise we'd never yield a pivot. Remember how I pointed out that quick sort missed out the pivot when specifying the sublists to recurse into? Well, that's relevant here. Fortunately, it's really easy to work out what to do. Knowing that everything up to and including "right" has been sorted means we just need to keep a cursor representing the next index to yield, and then just move that cursor up until it's positioned beyond "right". The code is probably easier to understand than the description:

```
int nextYield = 0;

var stack = new Stack<LeftRight>();
stack.Push(new LeftRight(0, count - 1));
while (stack.Count > 0)
{
    LeftRight leftRight = stack.Pop();
    int left = leftRight.left;
    int right = leftRight.right;
    if (right > left)
    {
        int pivot = left + (right - left) / 2;
        int pivotPosition = Partition(indexes, keys, left, right, pivot);
        // Push the right sublist first, so that we *pop* the
        // left sublist first
        stack.Push(new LeftRight(pivotPosition + 1, right));
        stack.Push(new LeftRight(left, pivotPosition - 1));
    }
    else
    {
        while (nextYield <= right)
        {
            yield return data[indexes[nextYield]];
            nextYield++;
        }
    }
}
```

Tada! It works (at least according to my tests).

I *have* tried optimizing this a little further, to deal with the case when right == left + 1, i.e. we're only sorting two elements. It feels like that *ought* to be cheaper to do explicitly than via pivoting and adding two pointless entries to the stack... but the code gets a *lot* more complicated (to the point where I had to fiddle significantly to get it working) and from what I've seen, it doesn't make much performance difference. Odd. If this were a production-quality library to be used in performance-critical situations I'd go further in the testing, but as it is, I'm happy to declare victory at this point.

Performance

So, how well does it perform? I've only performed crude tests, and they perplex me somewhat. I'm sure that last night, when I was running the "yield at the end" code, my tests were running twice as slowly in Edulinq as in LINQ to Objects. Fair enough - this is just a hobby, Microsoft have no doubt put a lot of performance testing effort into this. (That hasn't stopped them from messing up "descending" comparers, admittedly, as I found out last night to my amusement.) That was on my "meaty" laptop (which is 64-bit with a quad core i7). On my netbook this morning, the same

Edulinq code seemed to be running slightly faster than LINQ to Objects. Odd.

This evening, having pulled the "early out" code from the source repository, the Edulinq implementation is running faster than the LINQ to Objects implementation even when the "early out" isn't actually doing much good. That's just plain weird. I blame my benchmarking methodology, which is far from rigorous. I've tweaked the *parameters* of my tests quite a bit, but I haven't tried all kinds of different key and element types, etc. The basic results are very roughly:

- When evaluating the whole ordered list, Edulinq appears to run about 10% faster than LINQ to Objects
- When evaluating only the top 5 of a large ordered list, Edulinq can be *much* faster. How much faster depends on the size of the list of course, and it still has to perform the initial complete partitioning step - but on 100,000 items it's regularly about 10x faster than LINQ to Objects.

That makes me happy :) Of course, the code is all open source, so if Microsoft wish to include the Edulinq implementation in .NET 5, they're quite at liberty to do so, as long as they abide by the terms of the licence. I'm not holding my breath ;)

More seriously, I fully expect there are a bunch of scenarios where my knocked-up-in-an-evening code performs slower than that in the framework. Maybe my approach takes a lot more memory. Maybe it has worse locality of reference in some scenarios. There are all kinds of possibilities here. Full performance analysis was never meant to be the goal of Edulinq. I'm doing this in the spirit of learning more about LINQ - but it's fun to try to optimize just a *little* bit. I'm going to delete the increasingly-inaccurately-named MergeSortTest project now - I may institute a few more benchmarks later on though. I'm also removing CompoundComparer and ProjectionComparer, which are no longer used. They'll live on in part 26a though...

Conclusions

Well that was fun, wasn't it? I'm pretty pleased with the result. The final code has some nasty generic complexity in it, but it's not too bad if you keep all the types clear in your mind.

None of the remaining operators will be nearly as complex as this, unless I choose to implement AsQueryable (which I wasn't planning on doing). On the other hand, as I've mentioned before, Max/Sum/etc have *oodles* of overloads. While I'll certainly implement all of them, I'm sure I'll only present the *code* for selected interesting overloads.

As a bit of light relief, I think I'll tackle Reverse. That's about as simple as it gets -

although it could still present some interesting options.

Addendum

An earlier version of this post (and the merge sort implementation) had a flawed piece of code for choosing the pivot. Here's both the old and the new code:

```
// Old code
int pivot = (left + right) / 2;

// New code
int pivot = left + (right - left) / 2;
```

The difference is whether or not the code can overflow when left and right are very large. Josh Bloch wrote about it back in 2006. A colleague alerted me to this problem shortly after posting, but it's taken until now to correct it. (I fixed the source repository almost immediately, but deferred writing this addendum.) Why was I not too worried? Because .NET restricts each object to be less than 2GB in size, even in .NET 4.0, even on a 64-bit CLR. As we've created an array of integers, one per entry, that means we can only have just under (int.MaxValue / 4) elements. Within those limits, there's no problem in the original pivot code. However, it's still worth fixing of course - one never knows when the restriction will be lifted. The CLR team blogged about the issue back in 2005 (when the 64-bit CLR was new) - I haven't seen any mentions of plans to remove the limitation, but I would imagine it's discussed periodically.

One oddity about this is that the Array class itself has some API support for large arrays, such as the LongLength property. To be honest, I can't see large arrays ever being particularly pleasant to work with - what would they return for the normal Length property, for example, or their implementation of IList<T> etc? I suspect we may see support for larger objects before we see support for arrays with more than int.MaxValue elements, but that's a complete guess.

Back to the table of contents.

Part 27 - Reverse

Time for a change of pace after the deep dive into sorting. Reversing is pretty simple... which is not to say there's nothing to discuss, of course.

What is it?

Reverse only has a single, simple signature:

```
public static IEnumerable<TSource> Reverse<TSource>(
    this IEnumerable<TSource> source)
```

The behaviour is pretty simple to describe:

- source cannot be null; this is validated eagerly.
- The operator uses deferred execution: until you start reading from the result, it won't read anything from the input sequence
- As soon as you start reading from the result sequence, the input sequence is read in its entirety
- The result sequence contains all the elements of the input sequence, in the opposite order. (So the first element of the result sequence is the last element of the input sequence.)

The third point is the most interesting. It sounds like an obvious requirement just to get it to work at all - until you think of possible optimizations. Imagine if you implemented Reverse with an optimization for arrays: we know the array won't change size, and we can find out that size easily enough - so we could just use the indexer on each iteration, starting off with an index of "length - 1" and decrementing until we'd yielded every value.

LINQ to Objects *doesn't* behave this way - and that's observable because if you change the value of the array after you start iterating over the result sequence, you don't see those changes. Deferred execution means that you *will* see changes made to the array after the call to Reverse but before you start iterating over the results, however.

Note that the buffering nature of this operator means that you can't use it on infinite sequences - which makes sense when you think about it. What's the last element of an infinite sequence?

What are we going to test?

Most of the tests are pretty obvious, but I have one test to demonstrate how the timing of changes to the contents of the input sequence affect the result sequence:

```
[Test]
public void ArraysAreBuffered()
{
    // A sneaky implementation may try to optimize for the case where the
collection
    // implements IList or (even more "reliable") is an array: it mustn't do
this,
    // as otherwise the results can be tainted by side-effects within
iteration
    int[] source = { 0, 1, 2, 3 };

    var query = source.Reverse();
    source[1] = 99; // This change *will* be seen due to deferred execution
    using (var iterator = query.GetEnumerator())
    {
        iterator.MoveNext();
        Assert.AreEqual(3, iterator.Current);

        source[2] = 100; // This change *won't* be seen
        iterator.MoveNext();
        Assert.AreEqual(2, iterator.Current);

        iterator.MoveNext();
        Assert.AreEqual(99, iterator.Current);

        iterator.MoveNext();
        Assert.AreEqual(0, iterator.Current);
    }
}
```

If you can think of any potentially-surprising tests, I'd be happy to implement them - there wasn't much I could think of in terms of corner cases.

Let's implement it!

Eager validation of source combined with deferred execution suggests the normal implementation of splitting the operator into two methods - I won't bother showing the public part, as it only does exactly what you'd expect it to. However, to make up for the fact that Reverse is so simple, I'll present three implementations of the "Impl" method.

First, let's use a collection which performs the reversing for us automatically: a stack. The iterator returned by Stack<T> returns items in the order in which they would be

seen by multiple calls to Pop - i.e. the reverse of the order in which they were added. This makes the implementation trivial:

```
private static IEnumerable<TSource> ReverseImpl<TSource>(IEnumerable<TSource>
source)
{
    Stack<TSource> stack = new Stack<TSource>(source);
    foreach (TSource item in stack)
    {
        yield return item;
    }
}
```

Again, with "yield foreach" we could have done this in a single statement.

Next up, a linked list. In some ways, using a linked list is very natural - you never need to resize an array, or anything like that. On the other hand, we have an extra node object for every single element, which is a massive overhead. It's not what I'd choose to use for production in this case, but it's worth showing:

```
private static IEnumerable<TSource> ReverseImpl<TSource>(IEnumerable<TSource>
source)
{
    LinkedList<TSource> list = new LinkedList<TSource>(source);
    LinkedListNode<TSource> node = list.Last; // Property, not method!
    while (node != null)
    {
        yield return node.Value;
        node = node.Previous;
    }
}
```

Finally, a more "close to the metal" approach using our existing ToBuffer method:

```
private static IEnumerable<TSource> ReverseImpl<TSource>(IEnumerable<TSource>
source)
{
    int count;
    TSource[] array = source.ToBuffer(out count);
    for (int i = count - 1; i >= 0; i--)
    {
        yield return array[i];
    }
}
```

This is probably not significantly more efficient than the version using Stack<T> - I

expect Stack<T> has a similar implementation to ToBuffer when it's constructed with an input sequence. However, as it's so easy to count down from the end of the array to the start, we don't really need to take advantage of any of the features of Stack - so we might as well just use the array directly.

Note that this relies on the fact that ToBuffer will create a copy of whatever it's given, including an array. That's okay though - we're relying on that all over the place :)

Conclusion

It's hard to see how this could really be optimized any further, other than by improving ToBuffer based on usage data. Overall, a lovely simple operator.

It's probably about time I tackled some of the arithmetic aggregation operators... so next time I'll probably implement Sum.

Back to the table of contents.

Part 28 - Sum

Okay, I've bitten the bullet. The first of the four Aggregation Operators Of Overload Doom (AOOOD) that I've implemented is Sum. It was far from *difficult* to implement - just tedious.

What is it?

Sum has 20 overloads - a set of 4 for each of the types that it covers (int, long, float, double, decimal). Here are the overloads for int:

```
public static int Sum(this IEnumerable<int> source)

public static int? Sum(this IEnumerable<int?> source)

public static int Sum<T>(
    this IEnumerable<T> source,
    Func<T, int> selector)

public static int? Sum<T>(
    this IEnumerable<T> source,
    Func<T, int?> selector)
```

As you can see, there are basically two variations:

- A source of the numeric type itself, or a source of an arbitrary type with a projection to the numeric type
- The numeric type can be nullable or non-nullable

The behaviour is as follows:

- All overloads use *immediate execution*: it will immediately iterate over the source sequence to compute the sum, which is obviously the return value.
- source and selector must both be non-null
- Where there's a selector, the operator is equivalent to source.Select(selector).Sum() - or you can think of the versions *without* a selector as using an identity selector
- Where the numeric type is nullable, null values are ignored
- The sum of an empty sequence is 0 (even for nullable numeric types)

The last point is interesting - because the overloads with nullable numeric types *never return a null value*. Initially I missed the fact that the return type even *was*

nullable. I think it's somewhat misleading to be nullable but never null - you might have at least expected that the return value for an empty sequence (or one consisting only of null values) would be null.

For int, long and decimal, overflow within the sum will throw OverflowException. For single and double, the result will be positive or negative infinity. If the sequence contains a "not-a-number" value (NaN), the result will be NaN too.

What are we going to test?

A lot!

In total, I have 123 tests across the five types. The tests are mostly the same for each type, with the exception of overflow behaviour and not-a-number behaviour for single and double. Each overload is tested reasonably thoroughly:

- Argument validation
- Summing a simple sequence
- Summing an empty sequence
- (Nullable) Summing a simple sequence containing null values
- (Nullable) Summing a sequence containing only null values
- Positive overflow (either to an exception or infinity)
- Negative overflow (only one test per type, rather than for each overload)
- (Single/Double) Sequences containing NaN values
- Projections resulting in all of the above

Most of this was done using cut and paste, leading to a 916-line source file. On Twitter, followers have suggested a couple of alternatives - templating (possibly for both the tests and the implementation), or using more advanced features of unit test frameworks. There's nothing wrong with these suggestions - but I'm always concerned about the balance within an elegant-but-complex solution to repetition. If it takes longer to get to the "neat" solution, and then each individual test is harder to read, is it worth it? It certainly makes it easier to add one test which is then applicable over several types, or to modify all "copies" of an existing test - but equally it makes it harder to make variations (such as overflow) fit within the pattern. I have no quarrel with the idea of using more advanced techniques here, but I've stuck to a primitive approach for the moment.

Let's implement it!

Again, I'll only demonstrate the "int" implementations - but talk about single/double later on. There are plenty of ways I *could* have implemented this:

- Delegate everything to the *simplest* overload using "Where" for the nullable sequences and "Select" for the projection sequences
- Delegate everything to the most *complex* overload using identity projections
- Implement each method independently
- Somewhere in-between :)

In the end I've implemented each non-projecting overload by delegating to the corresponding projection-based one with an identity projection. I've implemented the non-nullable and nullable versions separately though. Here's the complete implementation:

```
public static int Sum(this IEnumerable<int> source)
{
    return Sum(source, x => x);
}

public static int? Sum(this IEnumerable<int?> source)
{
    return Sum(source, x => x);
}

public static int Sum<T>(
    this IEnumerable<T> source,
    Func<T, int> selector)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    if (selector == null)
    {
        throw new ArgumentNullException("selector");
    }
    checked
    {
        int sum = 0;
        foreach (T item in source)
        {
            sum += selector(item);
        }
        return sum;
    }
}

public static int? Sum<T>(
    this IEnumerable<T> source,
    Func<T, int?> selector)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
```

```
    }
    if (selector == null)
    {
        throw new ArgumentNullException("selector");
    }
    checked
    {
        int sum = 0;
        foreach (T item in source)
        {
            sum += selector(item).GetValueOrDefault();
        }
        return sum;
    }
}
```

Note the use of Nullable<T>.GetValueOrDefault() to "ignore" null values - it felt easier to add zero than to use an "if" block here. I *suspect* it's also more efficient, as there's no need for any conditionality here: I'd expect the implementation of GetValueOrDefault() to just return the underlying "value" field within the Nullable<T>, without performing the check for HasValue which the Value property normally would.

Of course, if I were *really* bothered by performance I'd implement each operation separately, instead of using the identity projection.

Note the use of the "checked" block to make sure that overflow is handled appropriately. As I've mentioned before, it would quite possibly be a good idea to turn overflow checking on for the whole assembly, but here I feel it's worth making it explicit to show that we consider overflow as an important part of the behaviour of this operator. The single/double overloads *don't* use checked blocks, as their overflow behaviour isn't affected by the checked context.

Conclusion

One down, three to go! I suspect Min and Max will use even more cutting and pasting (with judiciously applied changes, of course). There are 22 overloads for each of those operators, due to the possibility of using an arbitrary type - but I may well be able to use the most generic form to implement *all* the numeric versions. I may measure the impact this has on performance before deciding for sure. Anyway, that's a topic for the next post...

Addendum

As has been pointed out in the comments, my original implementation used a float to accumulate values when summing a sequence of floats. This causes problems, as

these two new tests demonstrate:

```
[Test]
public void NonOverflowOfComputableSumSingle()
{
    float[] source = { float.MaxValue, float.MaxValue,
                       -float.MaxValue, -float.MaxValue };
    // In a world where we summed using a float accumulator, the
    // result would be infinity.
    Assert.AreEqual(0f, source.Sum());
}

[Test]
public void AccumulatorAccuracyForSingle()
{
    // 20000000 and 20000004 are both exactly representable as
    // float values, but 20000001 is not. Therefore if we use
    // a float accumulator, we'll end up with 20000000. However,
    // if we use a double accumulator, we'll get the right value.
    float[] array = { 20000000f, 1f, 1f, 1f, 1f };
    Assert.AreEqual(20000004f, array.Sum());
}
```

The second of these tests is specific to floating point arithmetic - there's no equivalent in the integer domain. Hopefully the comment makes the test clear. We could still do better if we used the Kahan summation algorithm, but I haven't implemented that yet, and don't currently intend to. Worth noting as a potential follow-on project though.

Back to the first test though: this certainly *can* be represented in integers. If we try to sum { int.MaxValue, int.MaxValue, -int.MaxValue, -int.MaxValue } there are two options: we can overflow (throwing an exception) or we can return 0. If we use a long accumulator, we'll return 0. If we use an int accumulator, we'll overflow. I genuinely didn't know what the result for LINQ to Objects would be until I tried it - and found that it overflows. I've added a test to document this behaviour:

```
[Test]
public void OverflowOfComputableSumInt32()
{
    int[] source = { int.MaxValue, 1, -1, -int.MaxValue };
    // In a world where we summed using a long accumulator, the
    // result would be 0.
    Assert.Throws<OverflowException>(() => source.Sum());
}
```

Of course, I could have gone my own way and made Edulinq more capable than LINQ to Objects here, but in this case I've gone with the existing behaviour.

Back to the table of contents.

Part 29 - Min/Max

The second and third AOOOD operators today... if I'm brave enough to tackle Average tomorrow, I'll have done them all. More surprises here today, this time in terms of documentation...

What are they?

Min and Max are both extension methods with 22 overloads *each*. Min looks like this:

```
public static int Min(this IEnumerable<int> source)

public static int Min<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int> selector)

public static int? Min(this IEnumerable<int?> source)

public static int? Min<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int?> selector)

// Repeat the above four overloads for long, float, double and decimal,
// then add two more generic ones:

public static TSource Min<TSource>(this IEnumerable<TSource> source)

public static TResult Min<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector
)
```

(Max is *exactly* the same as Min; just replace the name.)

The more obvious aspects of the behaviour are as follows:

- source and selector mustn't be null
- All overloads use immediate execution
- The minimum or maximum value within the sequence is returned
- If a selector is present, it is applied to each value within source, and the maximum of the *projected* values is returned. (Note how the return type of these methods is TResult, not TSource.)

Some less obvious aspects - in all cases referring to the result type (as the source

type is somewhat incidental when a selector is present; it doesn't affect the behaviour):

- The type's IComparable<T> implementation is used when available, otherwise IComparable is used. An ArgumentException is thrown if values can't be compared. Fortunately, this is exactly the behaviour of Comparer<T>.Default.
- For any nullable type (whether it's a reference type or a nullable value type), nulls within the sequence are ignored, and an empty sequence (or one which contains only null values) will cause a null value to be returned. If there are any non-null values in the sequence, the return value will be non-null. (Note that this is different from Sum, which will return the *non-null* zero value for empty sequences over nullable types.)
- For any non-nullable value type, an empty sequence will cause InvalidOperationException to be thrown.

The first point is particularly interesting when you consider the double and float types, and their "NaN" (not-a-number) values. For example, Math.Max regards NaN as greater than positive infinity, but Enumerable.Max regards positive infinity as being the greater of the two. Math.Min and Enumerable.Min agree, however, that NaN is less than negative infinity. (It would actually make sense to me for NaN to be treated as the numeric equivalent of null here, but that would be strange in other ways...) Basically, NaN behaves oddly in all kinds of ways. I believe that IEEE-754-2008 actually specifies behaviour with NaNs which encourages the results we're getting here, but I haven't verified that yet. (I can't find a free version of the standard online, which is troubling in itself. Ah well.)

The behaviour of the nullable and non-nullable types is well documented for the type-specific overloads using int, Nullable<int> etc. However, the generic overloads (the ones using TSource) are poorly documented:

- InvalidOperationException isn't in the list of possibly-thrown arguments for any of the overloads
- The methods using selectors from TSource to TResult don't mention the possibility of nullity at all
- The methods without selectors describe the behaviour of null values *for reference types*, but don't mention the possibility of empty sequences for non-nullable value types, or consider nullable value types at all.

(I should point out that ArgumentException isn't actually mentioned either for the case where values are incomparable, but that feels like a slightly less important offence for some reason. Possibly just because it didn't trip me up.)

If I remember, I'll open a Connect issue against this hole in the documentation when I

find time. Unlike the optimizations and set ordering (where it's reasonably forgivable to deliberately omit implementation details from the contract) you simply can't predict the behaviour in a useful way from the documentation here. And yes, I'm going on about this because it bit me. I had to resort to writing tests and running them against LINQ to Objects to see if they were correct or not. (They were incorrect in various places.)

If you look at the behaviour of the non-generic methods, the generic ones are entirely consistent of course.

There are a couple of things which you might consider "missing" in terms of Max and Min:

- The ability to find out the minimum/maximum value of a sequence *by* a projection. For example, consider a sequence of people. We may wish to find the youngest person in the sequence, in which case we'd like to be able to write something like:

  ```
  var oldest = people.MaxBy(person => person.Age);
  ```

  We can find the maximum age itself easily enough - but then we'd need a second pass to find the first person *with* that age. I've addressed this in MoreLINQ with the MaxBy and MinBy operators. The System.Interactive assembly in Reactive Extensions has the same methods too.
- The ability to specify a custom IComparer<T> implementation, as we can in most of the operators using IEqualityComparer<T>. For example, we can't find the "maximum" string in a sequence, using a case-insensitive ordinal comparison.

Still, at least that means there's less to test...

What are we going to test?

I decided I really couldn't find the energy to replicate all the tests for every type involved here. Instead, I have a bunch of tests for int and Nullable<int>, a few tests exploring the oddness of doubles, and a bunch of tests around the generic methods. In particular, I know that I've implemented decimal, float etc by calling the same methods that the int overloads use.

The tests cover:

- Argument validation
- Empty sequences

- Sequences of null values where applicable
- Projections of the above
- Generic tests for nullable and non-nullable value types, and reference types (with empty sequences etc)
- Incomparable values

Let's implement them!

Okay, let's start off with the simplest detail: the order of implementation:

- Max(int)
- Max(generic)
- Cut and paste Max implementations for other numeric types (replace the type name, basically)
- Cut and paste the entirety of Max to Min:
  - Replace "Max" with "Min" everywhere
  - Replace " < " with " > " everywhere (only 4 occurrences; basically the results of calling Compare or ComparerTo and comparing with 0)

Just as with Sum, I could have used templating - but I don't think it would actually have saved me significant time.

This time, I thought I'd use Select internally for the overloads with selectors (unlike my approach for Sum which used identity projections). There's no particular reason for this - I just thought it would be interesting to try both approaches. Overall, I think I prefer this one, but I haven't done any benchmarking to find out the relative performance penalties.

Each set of numeric overloads calls into a single pair of generic "implementation" methods. These aren't the public general-purpose ones: they require that the types in use implement IComparable<T>, and I've added a "struct" constraint just for kicks. This is just one approach. Other options:

- I could have implemented the code separately for each numeric type. That may well be faster than calling IComparable<T>.Compare (at least for most types) as the IL would have contained the appropriate operator directly. However, it would have meant more code and explicitly dealing with the headache of NaNs for double/float. If I ever write benchmarks, I'll investigate the difference that this can make.
- I could have used the public generic overloads, which eventually call into Comparer<T>.Default. Again, the penalty for this (if any) is unknown to me at this point. Can the JIT inline deeply enough to make this as fast as a "native" implementation? I wouldn't like to guess without tests.

I've separated out the nullable implementations from the non-nullable ones, as the behaviour differs significantly between the two.

Here's the public code for int:

```
public static int Max(this IEnumerable<int> source)
{
    return PrimitiveMax(source);
}

public static int Max<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int> selector)
{
    // Select will validate the arguments
    return PrimitiveMax(source.Select(selector));
}

public static int? Max(this IEnumerable<int?> source)
{
    return NullablePrimitiveMax(source);
}

public static int? Max<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int?> selector)
{
    // Select will validate the arguments
    return NullablePrimitiveMax(source.Select(selector));
}
```

All the methods consider argument validation to be somebody else's problem - either Select or the generic method we're calling to find the maximum value. Part of me thinks this is lazy; part of me likes it in terms of not repeating code. All of me would prefer the ability to specify non-nullable parameters declaratively...

Here are the "primitive" methods called into above:

```
// These are uses by all the overloads which use a known numeric type.
// The term "primitive" isn't truly accurate here as decimal is not a
primitive
// type, but it captures the aim reasonably well.
// The constraint of being a value type isn't really required, because we
don't rely on
// it within the method and only code which already knows it's a comparable
value type
// will call these methods anyway.

private static T PrimitiveMax<T>(IEnumerable<T> source) where T : struct,
```

```
IComparable<T>
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    using (IEnumerator<T> iterator = source.GetEnumerator())
    {
        if (!iterator.MoveNext())
        {
            throw new InvalidOperationException("Sequence was empty");
        }
        T max = iterator.Current;
        while (iterator.MoveNext())
        {
            T item = iterator.Current;
            if (max.CompareTo(item) < 0)
            {
                max = item;
            }
        }
        return max;
    }
}

private static T? NullablePrimitiveMax<T>(IEnumerable<T?> source) where T :
struct, IComparable<T>
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    T? max = null;
    foreach (T? item in source)
    {
        if (item != null &&
            (max == null || max.Value.CompareTo(item.Value) < 0))
        {
            max = item;
        }
    }
    return max;
}
```

The first method is interesting in terms of its approach to throwing an exception if the first element isn't present, and using that as an initial candidate otherwise.

The second method needs to consider nullity twice on each iteration:

- Is the item from the sequence null? If so, we can ignore it.
- Is our "current maximum" null? If so, we can replace it with the item from the sequence without performing a comparison.

Now there's one case which is ambiguous here: when *both* values are null. At that point we can choose to replace our "current maximum" with the item, or not... it doesn't matter as the values are the same anyway. It *is* important that we don't try to perform a comparison unless both values are non-null though... the short-circuiting && and || operators keep us safe here.

Having implemented the code above, all the *interesting* work lies in the generic forms. Here we don't have different public methods to determine which kind of behaviour we'll use: but I wrote two *private* methods instead, and just delegated to the right one from the public one. This seemed cleaner than putting the code all in one method:

```
public static TSource Max<TSource>(
    this IEnumerable<TSource> source)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    // This condition will be true for reference types and nullable value
types, and false for
    // non-nullable value types.
    return default(TSource) == null ? NullableGenericMax(source) :
NonNullableGenericMax(source);
}

public static TResult Max<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector)
{
    return Max(source.Select(selector));
}

/// <summary>
/// Implements the generic behaviour for non-nullable value types.
/// </summary>
/// <remarks>
/// Empty sequences will cause an InvalidOperationException to be thrown.
/// Note that there's no *compile-time* validation in the caller that the
type
/// is a non-nullable value type, hence the lack of a constraint on T.
/// </remarks>
private static T NonNullableGenericMax<T>(IEnumerable<T> source)
{
    Comparer<T> comparer = Comparer<T>.Default;

    using (IEnumerator<T> iterator = source.GetEnumerator())
    {
        if (!iterator.MoveNext())
```

```
        {
            throw new InvalidOperationException("Sequence was empty");
        }
        T max = iterator.Current;
        while (iterator.MoveNext())
        {
            T item = iterator.Current;
            if (comparer.Compare(max, item) < 0)
            {
                max = item;
            }
        }
        return max;
    }
}

/// <summary>
/// Implements the generic behaviour for nullable types - both reference
types and nullable
/// value types.
/// </summary>
/// <remarks>
/// Empty sequences and sequences comprising only of null values will cause
the null value
/// to be returned. Any sequence containing non-null values will return a
non-null value.
/// </remarks>
private static T NullableGenericMax<T>(IEnumerable<T> source)
{
    Comparer<T> comparer = Comparer<T>.Default;

    T max = default(T);
    foreach (T item in source)
    {
        if (item != null &&
            (max == null || comparer.Compare(max, item) < 0))
        {
            max = item;
        }
    }
    return max;
}
```

As you can tell, there's a significant similarity between the "PrimitiveMax" and "NonNullableGenericMax" methods, and likewise between "NullablePrimitiveMax" and "NullableGenericMax". This should come as no surprise. Fundamentally the difference is just between using an IComparable<T> implementation, and using Comparer<T>.Default. (The argument validation occurs in a different place too, as we'll be going through a public entry point for the non-primitive code.)

Once I'd discovered the correct behaviour, this was reasonably simple. Of course,

the above code wasn't my *first* implementation, where I'd completely forgotten about null values, and hadn't thought about how the nullability of the source type might affect the behaviour of empty sequences...

Conclusion

If you're ever in a company which rewards you for checking in lots of lines of code, offer to implement Sum/Min/Max. This weekend I've checked in about 2,500 lines of code in (split between production and test) and none of it's been terribly hard. Of course, if you're ever in such a company you should *also* consider looking for another job. (Have I mentioned that Google's hiring? Email me if you're interested. I'm serious.)

As you can tell, I was slightly irritated by the lack of clarity around the documentation in some places - but I find it interesting that even a simple-sounding function like "find the maximum value from a sequence" should *need* the kind of documentation that's missing here. I'm not saying it's a failure of the design - more just musing how a complete specification is almost always going to be longer than you might think at first glance. And if you think I was diligent here, think again: I didn't bother specifying *which* maximum or minimum value would be returned if there were two. For example, if a sequence consists of references to two equal but distinct strings, which reference should be returned? I have neither stated what my implementation (or the LINQ to Objects implementation) will do, nor tested for it.

Next up is Average - a single method with a mere 20 overloads. There are various corner cases to consider... but that's a post for another day.

Back to the table of contents.

Part 30 - Average

This is the final aggregation operator, after which I suspect we won't need to worry about floating point difficulties any more. Between this and the unexpected behaviour of Comparer<string>.Default, I've covered two of my "big three" pain points. It's hard to see how I could get dates and times into Edulinq naturally; it's even harder to see how time zones could cause problems. I've still got a few operators to go though, so you never know...

What is it?

Average has 20 overloads, all like the following but for long, decimal, float and double as well as int:

```
public static double Average(this IEnumerable<int> source)

public static double Average<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int> selector)

public static double? Average(this IEnumerable<int?> source)

public static double? Average<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int?> selector)
```

The operators acting on float sequences return float, and likewise the operators acting on decimal sequences return decimal, with the same equivalent nullable types for the nullable sequences.

As before (for Min/Max/Sum), the overloads which take a selector are equivalent to just applying that selector to each element in the sequence.

General behaviour - pretty much as you'd expect, I suspect:

- Each operator calculates the arithmetic mean of a sequence of values.
- source and selector can't be null, and are validated immediately.
- The operators all use *immediate execution*.
- The operators all iterate over the entire input sequence, unless an exception is thrown (e.g. due to overflow).
- The operators with a non-nullable return type throw InvalidOperationException if the input sequence is empty.

- The operators with a nullable return type ignore any null input values, and return null if the input sequence is empty or contains only null values. If non-null values are present, the return value will be non-null.

It all sounds pretty simple, doesn't it? We just sum the numbers, and divide by the count. It's not *too* complicated, but we have a couple of things to consider:

- How should we count items - which data type should we use? Do we need to cope with more than int.MaxValue elements?
- How should we sum items? Should we be able to find the average of { int.MaxValue, int.MaxValue } even though the sum clearly overflows the bounds of int?

Given the behaviour of my tests, I believe I've made the same decisions. I use a long for the counter, always. I use a long total for the int/long overloads, a double total for the float/double overloads, and a decimal total for the decimal overloads. These aren't particularly tricky decisions once you'd realised that you need to make them, but it would be very easy to implement the operators in a simplistic way without thinking about such things. (I'd probably have done so if it weren't for the comments around Sum this morning.)

What are we going to test?

I've only got in-depth tests for the int overloads, covering:

- Argument validation
- Empty sequences for nullable and non-nullable types
- Sequences with only null values
- Sequences of perfectly normal values :)
- Projections for all the above
- A sequence with just over int.MaxValue elements, to test we can count properly

Then I have a few extra tests for interesting situations. First I check the overflow behaviour of each type, using a common pattern of averaging a sequence of (max, max, -max, -max) where "max" is the maximum value for the sequence type. The results are:

- For int we get the correct result of 0 because we're accumulating over longs
- For long we get an OverflowException when it tries to add the first two values together
- For float we get the correct result of 0 because we're accumulating over doubles
- For double we get PositiveInfinity because that's the result of the first addition
- For decimal we get an OverflowException when it tries to add the first two

values together

Additionally, I have a couple of floating-point-specific tests: namely further proof that we use a double accumulator when averaging floats, and the behaviour of Average in the presence of NaN values:

```
[Test]
public void SingleUsesDoubleAccumulator()
{
    // All the values in the array are exactly representable as floats,
    // as is the correct average... but intermediate totals aren't.
    float[] array = { 20000000f, 1f, 1f, 2f };
    Assert.AreEqual(5000001f, array.Average());
}

[Test]
public void SequenceContainingNan()
{
    double[] array = { 1, 2, 3, double.NaN, 4, 5, 6 };
    Assert.IsNaN(array.Average());
}
```

I'm sure someone can think of some other interesting scenarios I should be considering :)

Let's implement it!

This is another cut-and-paste job, but with more editing required - for each method, I needed to make sure I was using the right accumulator type, and I occasionally removed redundant casts. Still, the code follows pretty much the same pattern for all types. Here's the int implementation:

```
public static double Average(this IEnumerable<int> source)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    checked
    {
        long count = 0;
        long total = 0;
        foreach (int item in source)
        {
            total += item;
            count++;
        }
        if (count == 0)
```

```
        {
            throw new InvalidOperationException("Sequence was empty");
        }
        return (double)total / (double)count;
    }
}

public static double Average<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int> selector)
{
    return source.Select(selector).Average();
}

public static double? Average(this IEnumerable<int?> source)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    checked
    {
        long count = 0;
        long total = 0;
        foreach (int? item in source)
        {
            if (item != null)
            {
                count++;
                total += item.Value;
            }
        }
        return count == 0 ? (double?)null : (double)total / (double)count;
    }
}

public static double? Average<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int?> selector)
{
    return source.Select(selector).Average();
}
```

Salient points:

- Again I'm using Select to make the implementation of the overloads with selectors trivial
- I've cast *both* operands of the division when calculating the average, just for clarity. We could get away with either of them.
- In the case of the conditional operator, I could actually *just* cast one of the division operators to "double?" and then remove *both* of the other casts...

again, I feel this version is clearer. (I could change my mind tomorrow, mind you...)
- I've explicitly used checked blocks for int and long. For float and double we won't get overflow anyway, and for decimal the checked/unchecked context is irrelevant.

There's one optimization we can perform here. Consider this loop, for the nullable sequence:

```
long count = 0;
long total = 0;
foreach (int? item in source)
{
    if (item != null)
    {
        count++;
        total += item.Value; // This line can be optimized...
    }
}
```

The line I've highlighted seems perfectly reasonable, right? We're trying to add the "real" non-null value within a value type value, and we know that there *is* a real value, because we've checked it's not the null value already.

Now think about what the Value property actually does... it checks whether or not it's the null value, and then returns the real value or throws an exception. But we *know* it won't throw an exception, because we've checked it. We just want to get at the value - don't bother with any more checks. *That's exactly what GetValueOrDefault() does.* In the case where the value is non-null, GetValueOrDefault() and the Value property obviously do the same thing - but intuition tells me that GetValueOrDefault() can do it quicker, because it doesn't actually need to check anything. It can just return the value of the underlying field - which will be the default value of the underlying type for a null wrapper value anyway.

I've benchmarked this, and on my laptop it's about 5% faster than using Value. But... it's such a grotty hack. I would feel dirty putting it in. Surely Value is the more readable code here - it just happens to be slower. As always, I'm undecided. There's no behavioural difference, just a slight speed boost. Thoughts, folks?

Conclusion

I'm quite pleased to be shot of the Aggregate Operators Of Overload Doom. I've felt for a while that they've been hanging over me - I knew they'd be annoying in terms of cut and paste, but there's been more interesting situations to consider than I'd

expected.

There's not a lot left now. According to my previous list, I've got:

- Cast and OfType
- ElementAt and ElementAtOrDefault
- SequenceEqual
- Zip (from .NET 4)
- Contains

However, that doesn't include AsEnumerable and AsQueryable. I'm unsure at the moment what I'm doing with those... AsEnumerable is trivial, and probably worth doing... AsQueryable could prove interesting in terms of testing, as it requires expression trees (which are in System.Core; a library I'm not referencing from tests when testing the Edulinq implementation). I'll play around and see what happens :)

Not sure what I'll implement next, to be honest... we'll see tomorrow!

Back to the table of contents.

Part 31 - ElementAt / ElementAtOrDefault

A nice easy pair of operators tonight. I should possibly have covered them at the same time as First/Last/Single and the OrDefault variants, but never mind...

What are they?

ElementAt and ElementAtOrDefault have a single overload each:

```
public static TSource ElementAt<TSource>(
    this IEnumerable<TSource> source,
    int index)

public static TSource ElementAtOrDefault<TSource>(
    this IEnumerable<TSource> source,
    int index)
```

Isn't that blissfully simple after the overload storm of the past few days?

The two operators work in very similar ways:

- They use *immediate execution*.
- The source parameter must not be null, and this is validated immediately.
- They return the element at the specified zero-based index, if it's in the range 0 <= index < count.

The methods only differ in their handling of an index which falls outside the given bound. ElementAt will throw an ArgumentOutOfRangeException; ElementAtOrDefault will return the default value for TSource (e.g. 0, null, false). This is true even if index is negative. You might have expected some way to specify the default value to return if the index is out of bounds, but there isn't one. (This is consistent with FirstOrDefault() and so on, but not with Nullable<T>.GetValueOrDefault())

This behaviour leaves us some room for common code - for once I haven't used cut and paste for the implementation. Anyway, I'm getting ahead of myself.

What are we going to test?

As you can imagine, my tests for the two operators are identical except for the expected result in the case of the index being out of range. I've tested the following

cases:

- Null source
- A negative index
- An index which is too big on a NonEnumerableCollection
- An index which is too big on a NonEnumerableList
- An index which is too big on a lazy sequence (using Enumerable.Range)
- A valid index in a NonEnumerableList
- A valid index in a lazy sequence

The "non-enumerable" list and collection are to test that the optimizations we're going to perform are working. In fact, the NonEnumerableCollection test fails on LINQ to Objects - it's only optimized for IList<T>. You'll see what I mean in a minute... and why that might not be a bad thing.

None of the tests are very interesting, to be honest.

Let's implement it!

As I mentioned earlier, I've used some common code for once (although I admit the first implementation used cut and paste). As the only difference between the two methods is the handling of a particular kind of failure, I've used the TryXXX pattern which exists elsewhere in the framework. There's a common method which *tries* to retrieve the right element as an out parameter, and indicates whether or not it succeeded via the return value. Not every kind of failure is just returned, of course - we want to throw an ArgumentNullException if source is null in either case.

That leaves our public methods looking quite straightforward:

```
public static TSource ElementAt<TSource>(
    this IEnumerable<TSource> source,
    int index)
{
    TSource ret;
    if (!TryElementAt(source, index, out ret))
    {
        throw new ArgumentOutOfRangeException("index");
    }
    return ret;
}

public static TSource ElementAtOrDefault<TSource>(
    this IEnumerable<TSource> source,
    int index)
{
    TSource ret;
```

```
    // We don't care about the return value - ret will be default(TSource) if
it's false
    TryElementAt(source, index, out ret);
    return ret;
}
```

TryElementAt will only return false if the index is out of bounds, so the exception is always appropriate. However, there is a disadvantage to this approach: we can't easily indicate in the exception message whether index was too large or negative. We could have specified a message which included the value of index itself, of course. I think it's a minor matter either way, to be honest.

The main body of the code is in TryElementAt, obviously. It would actually be very simple - just looping and counting up to index, checking as we went - except there are two potential optimizations.

The most obvious - and most profitable - optimization is if the collection implements IList<T>. If it does, we can efficiently obtain the count using the ICollection<T>.Count property (don't forget that IList<T> extends ICollection<T>), check that it's not too big, and then use the indexer from IList<T> to get straight to the right element. Brilliant! That's a clear win.

The less clear optimization is if the collection implements ICollection<T> but not IList<T>, or if it only implements the nongeneric ICollection. In those cases we can still get at the count - but we can't then get directly to the right element. In other words, we can optimize the *failure* case (possibly hugely), but at a very slight cost - the cost of checking whether the sequence implements either interface - for the success case, where the check won't do us any good.

This is the sort of optimization which is impossible to judge without real data. How often are these operators called with an invalid index? How often does that happen on a collection which implements ICollection<T> but not IList<T> (or implements ICollection)? How large are those collections (so how long would it take to have found our error the normal way)? What's the cost of performing the type check? I don't have the answers to *any* of these questions. I don't even have strong suspicions. I know that Microsoft doesn't use the same optimization, but I don't know whether that was due to hard data or a gut feeling.

For the moment, I've kept all the optimizations. Here's the code:

```
private static bool TryElementAt<TSource>(
    IEnumerable<TSource> source,
    int index,
    out TSource element)
```

```
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    element = default(TSource);
    if (index < 0)
    {
        return false;
    }
    ICollection<TSource> collection = source as ICollection<TSource>;
    if (collection != null)
    {
        int count = collection.Count;
        if (index >= count)
        {
            return false;
        }
        // If it's a list, we know we're okay now - just return directly...
        IList<TSource> list = source as IList<TSource>;
        if (list != null)
        {
            element = list[index];
            return true;
        }
    }

    ICollection nonGenericCollection = source as ICollection;
    if (nonGenericCollection != null)
    {
        int count = nonGenericCollection.Count;
        if (index >= count)
        {
            return false;
        }
    }
    // We don't need to fetch the current value each time - get to the right
    // place first.
    using (IEnumerator<TSource> iterator = source.GetEnumerator())
    {
        // Note use of -1 so that we start off my moving onto element 0.
        // Don't want to use i <= index in case index == int.MaxValue!
        for (int i = -1; i < index; i++)
        {
            if (!iterator.MoveNext())
            {
                return false;
            }
        }
        element = iterator.Current;
        return true;
    }
}
```

As you can see, the optimized cases actually form the bulk of the code - part of me thinks it would be worth removing the non-IList<T> optimizations just for clarity and brevity.

It's worth looking at the "slow" case where we actually iterate. The for loop looks odd, until you think that to get at element 0, you have to call MoveNext() once. We don't want to just add one to index or use a less-than-or-equal condition: both of those would fail in the case where index is int.MaxValue; we'd either not loop at all (by incrementing index and it overflowing either causing an exception or becoming negative) or we'd loop forever, as *every* int is less than or equal to int.MaxValue.

Another way to look at it is that the loop counter ("i") is the "current index" within the iterator: the iterator starts before the first element, so it's reasonable to start at -1.

The reason I'm drawing attention to this is that I got all of this wrong first time... and was very grateful for unit tests to catch me out.

Conclusion

For me, the most interesting part of ElementAt is the decision about optimization. I'm sure I'm not the only one who optimizes without data at times - but it's a dangerous thing to do. The problem is that this isn't the normal micro-optimization quandary of "it's always a tiny bit better, but it's probably insignificant and makes the code harder to read". For the cases where this is faster, it could make an enormous difference - asking for element one million of a linked list which doesn't *quite* have enough elements could be very painful. But do failure cases need to be fast? How common are they? As you can tell, I'm dithering. I think it's at least worth thinking about what optimizations *might* make a difference - even if we later remove them.

Next time, I think I'll tackle Contains - an operator which you might expect to be really fast on a HashSet<T>, but which has some interesting problems of its own...

Back to the table of contents.

Part 32 - Contains

After the dubious optimizations of ElementAt/ElementAtOrDefault yesterday, we meet an operator which is remarkably good at defying optimization. Sort of. Depending on how you feel it should behave.

What is it?

Contains has two overloads, which only differ by whether or not they take an equality comparer - just like Distinct, Intersect and the like:

```
public static bool Contains<TSource>(
    this IEnumerable<TSource> source,
    TSource value)

public static bool Contains<TSource>(
    this IEnumerable<TSource> source,
    TSource value,
    IEqualityComparer<TSource> comparer)
```

The operator simply returns a Boolean indicating whether or not "value" was found in "source". The salient points of its behaviour should be predictable now:

- It uses immediate execution (as it's returning a simple value instead of a sequence)
- The source parameter cannot be null, and is validated immediately
- The value parameter *can* be null: it's valid to search for a null value within a sequence
- The comparer parameter can be null, which is equivalent to passing in EquailtyComparer<TSource>.Default.
- The overload without a comparer uses the default equality comparer too.
- If a match is found, the method returns immediately without reading the rest of the input sequence.
- There's a documented optimization for ICollection<T> - but there are significant issues with it...

So far, so good.

What are we going to test?

Aside from argument validation, I have tests for the value being present in the source,

and it *not* being present in the source... for the three options of "no comparer", "null comparer" and "specific comparer".

I then have one final test to validate that we return as soon as we've found a match, by giving a query which will blow up when the element after the match is computed.

Frankly none of the tests are earth-shattering, but in the spirit of giving you an idea of what they're like, here's one with a custom comparer - we use the same source and value for a "default comparer" test which *doesn't* find the value as the case differs:

```
[Test]
public void MatchWithCustomComparer()
{
    // Default equality comparer is ordinal
    string[] source = { "foo", "bar", "baz" };
    Assert.IsTrue(source.Contains("BAR", StringComparer.OrdinalIgnoreCase));
}
```

Currently I *don't* have a test for the optimization mentioned in the bullet points above, as I believe it's broken. More later.

Let's implement it!

To start with, let's dispense with the overload without a comparer parameter: that just delegates to the other one by specifying EqualityComparer<TSource>.Default. Trivial. (Or so we might think. There's more to this than meets the eye.)

I've got three implementations, but we'll start with just two of them. Which one you pick would depend on whether you're happy to use one operator to implement another. If you think that's okay, it's really simple:

```
public static bool Contains<TSource>(
    this IEnumerable<TSource> source,
    TSource value,
    IEqualityComparer<TSource> comparer)
{
    comparer = comparer ?? EqualityComparer<TSource>.Default;
    return source.Any(item => comparer.Equals(value, item));
}
```

"Any" has exactly the traits we want, including validation of the non-nullity of "source". It's hardly complicated code if we *don't* use Any though:

```
public static bool Contains<TSource>(
```

```
    this IEnumerable<TSource> source,
    TSource value,
    IEqualityComparer<TSource> comparer)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    comparer = comparer ?? EqualityComparer<TSource>.Default;
    foreach (TSource item in source)
    {
        if (comparer.Equals(value, item))
        {
            return true;
        }
    }
    return false;
}
```

Obviously there's a *slight* penalty in using Any just because of executing a delegate on each iteration - and the extra memory requirement of building an object to capture the comparer. I haven't measured the performance impact of this - again, it's a candidate for benchmarking.

Can't we optimize? (And why does LINQ to Objects think it can?)

The implementations above are all very well, but they feel ever so simplistic. With ElementAt, we were able to take advantage of the fact that an IList<T> allows us random access by index. Surely we've got similar collections which allow us to test for containment cheaply?

Well, yes and no. We've got IDictionary<TKey, TValue> which allows you to check for the presence of a particular *key* - but even it would be hard to test whether the sequence we're looking at is the key sequence for some IDictionary<TSource, TValue>, and somehow get back to the dictionary.

ICollection<T> has a Contains method, but that doesn't necessarily do the right thing. This is particularly troubling, as the MSDN documentation for the comparer-less overload has contradictory information:

> (Summary)
>
> Determines whether a sequence contains a specified element by using the default equality comparer.
>
> (Remarks)

If the type of source implements ICollection<T>, the **Contains** method in that implementation is invoked to obtain the result. Otherwise, this method determines whether source contains the specified element.

Enumeration is terminated as soon as a matching element is found.

Elements are compared to the specified value by using the default equality comparer, Default.

Why is this troubling? Well, let's look at a test:

```
[Test]
public void SetWithDifferentComparer()
{
    HashSet<string> sourceAsSet = new
HashSet<string>(StringComparer.OrdinalIgnoreCase)
        { "foo", "bar", "baz" };
    IEnumerable<string> sourceAsSequence = sourceAsSet;
    Assert.IsTrue(sourceAsSet.Contains("BAR"));
    Assert.IsFalse(sourceAsSequence.Contains("BAR"));
    Assert.IsFalse(sourceAsSequence.Contains("BAR", StringComparer.Ordinal));
}
```

(This exact code won't build in the Edulinq project configuration, as that doesn't have a reference to the System.Core assembly which contains HashSet<T>. I've got a hack which allows me to run *effectively* this code though. See the source for details.)

Now this test looks correct to me: while we're regarding the set *as* a set, it should use the set's comparer and find "BAR" with a case-insensitive match. However, when we use it as a *sequence* in LINQ, it should obey the rules of Enumerable.Contains - which means that the middle call should use the default equality comparer for string. Under that equality comparer, "BAR" *isn't* present.

It doesn't: the above test fails on that middle call in LINQ to Objects, because HashSet<T> implements ICollection<T>. To fit in with the implementation, the documentation summary should actually be worded as something like:

"Determines whether a sequence contains a specified element by using the default equality comparer if the sequence doesn't implement ICollection<T>, or whatever equality comparison the collection uses if it *does* implement ICollection<T>."

Now you may be saying to yourself that this is only like relying on IList<T> to fetch an item by index in a fashion consistent with iterating over with it - but I'd argue that any IList<T> implementation which *didn't* do that was simply broken... whereas ICollection<T>.Contains is specifically documented to allow custom comparisons:

> Implementations can vary in how they determine equality of objects; for example, List<T> uses Comparer<T>.Default, whereas Dictionary<TKey, TValue> allows the user to specify the IComparer<T> implementation to use for comparing keys.

Let's leave aside the fact that those "Comparer<T>" and "IComparer<T>" should be "EqualityComparer<T>" and "IEqualityComparer<T>" respectively for the minute, and just note that it's entirely reasonable for an implementation *not* to use the default equality comparer. That makes sense - but I believe it also makes sense for source.Contains(value) to be more predictable in terms of the equality comparer it uses.

Now I would certainly agree that having a method call which changes semantics based on whether the compile-time type of the source is IEnumerable<T> or ICollection<T> is undesirable too... but I'm not sure there *is* any particularly nice solution. The options are:

- The current LINQ to Objects implementation where the comparer used is hard to predict.
- The Edulinq implementation where the type's default comparer is always used... *if* the compile-time type means that Enumerable.Contains is used in the first place.
- Remove the comparer-less overload entirely, and force people to specify one. This is lousy for convenience.

Note that you might expect the overload which takes a comparer to work the same way if you pass in null as the comparer - but it doesn't. That overload *never* delegates to ICollection<T>.Contains.

So: convenience, predictability, consistency. Pick any two. Isn't API design fun? This isn't even thinking about performance, of course...

It's worth bearing in mind that even the current behaviour which is presumably meant to encourage consistency doesn't work. One might expect that the following would always be equivalent for any sensible collection:

```
var first = source.Contains(value);
var second = source.Select(x => x).Contains(value);
```

... but of course the second line will always use EqualityComparer<T>.Default whereas the first may or may not.

(Just for fun, think about Dictionary<TKey, TValue> which implements

ICollection<KeyValuePair<TKey, TValue>>; its explicitly-implemented ICollection<T>.Contains method will use its own equality comparer for the key, but the default equality comparer for the value part of the pair. Yay!)

So can we *really* not optimize?

I can think of exactly one situation which we could legitimately optimize without making the behaviour hard to predict. Basically we're fine to ask the collection to do our work for us if we can guarantee it will use the right comparer. Ironically, List<T>.Contains has an overload which allows us to specify the equality comparer, so we *could* delegate to that - but it's not going to be significantly faster than doing it ourselves. It's still got to look through everything.

ISet<T> in .NET 4 doesn't help us much - its API doesn't talk about equality comparers. (This makes a certain amount of sense - consider SortedSet<T> which uses IComparer<T> instead of IEqualityComparer<T>. It wouldn't make sense to ask a SortedSet<T> for an equality comparer - it couldn't give you one, as it wouldn't know how to produce a hash code.)

However, HashSet<T> *does* give us something to work with. You can *ask* a HashSet<T> which equality comparer it uses, so we could delegate to its implementation *if and only if it would use the one we're interested in*. We can bolt that into our existing implementation pretty easily, after we've worked out the comparer to use:

```
HashSet<TSource> hashSet = source as HashSet<TSource>;
if (hashSet != null && comparer.Equals(hashSet.Comparer))
{
    return hashSet.Contains(value);
}
```

So is this worth including or not?

Pros:

- It covers one of the biggest use cases for optimizing Contains; I *suspect* this is used more often than the LINQ implementation of Contains working over a dictionary.
- So long as the comparer doesn't override Equals in a bizarre way, it should be a true optimization with no difference in behaviour.
- The optimization is applied for *both* overloads of Enumerable.Contains, not just the comparer-less one.

Cons:

- It's specific to HashSet<T> rather than an interface type. That makes it feel a little too specific to be a good target of optimization.
- We've still got the issue of consistency in terms of sourceAsSet.Contains(value) vs sourceAsSequence.Contains(value)
- There's a tiny bit of overhead if the source isn't a hash set, and a further overhead if it *is* a hash set but with the wrong comparer. I'm not too bothered about this.

It's not the default implementation in Edulinq at the moment, but I could possibly be persuaded to include it. Likewise I have a conditionally-compiled version of Contains which *is* compatible with LINQ to Objects, with the "broken" optimization for the comparer-less overload; this is turned off by default too.

Conclusion

Gosh! I hadn't expected Contains to be *nearly* this interesting. I'd worked out that optimization would be a pain, but I hadn't expected it to be such a weird design choice.

This is the first time I've deliberately gone against the LINQ to Objects behaviour, other than the MS bug around descending orderings using "extreme" comparers. The option for compatibility is there, but I feel fairly strongly that this was a bad design decision on Microsoft's part. A bad decision out of some fairly unpleasant alternatives, I grant you. I'm willing to be persuaded of its virtues, of course - and in particular I'd welcome discussion with the LINQ team around this. In particular, it's always fun to hear about the history of design decisions.

Next up, Cast and OfType.

Back to the table of contents.

Part 33 - Cast and OfType

More design decisions around optimization today, but possibly less controversial ones...

What are they?

Cast and OfType are somewhat unusual LINQ operators. They are extension methods, but they work on the non-generic IEnumerable type instead of the generic IEnumerable<T> type:

```
public static IEnumerable<TResult> Cast<TResult>(this IEnumerable source)

public static IEnumerable<TResult> OfType<TResult>(this IEnumerable source)
```

It's worth mentioning what Cast and OfType are used for to start with. There are two main purposes:

- Using a non-generic collection (such as a DataTable or an ArrayList) within a LINQ query (DataTable has the AsEnumerable extension method too)
- Changing the type of a generic collection, usually to use a more specific type (e.g. you have List<Person> but you're confident they're all actually Employee instances - or you only want to query against the Employee instances)

I can't say that I use either operator terribly often, but if you're starting off from a non-generic collection for whatever reason, these two are your only easy way to get "into" the LINQ world.

Here's a quick rundown of the behaviour they have in common:

- The source parameter must not be null, and this is validated eagerly
- It uses *deferred execution*: the input sequence is not read until the output sequence is
- It streams its data - you can use it on arbitrarily-long sequences and the extra memory required will be constant (and small :)

Both operators effectively try to convert each element of the input sequence to the result type (TResult). When they're successful, the results are equivalent (ignoring optimizations, which I'll come to later). The operators differ in how they handle elements which aren't of the result type.

Cast simply tries to cast each element to the result type. If the cast fails, it will throw an InvalidCastException in the normal way. OfType, however, sees whether each element *is* a value of the result type first - and ignores it if it's not.

There's one important case to consider where Cast will successfully return a value and OfType will ignore it: null references (with a nullable return type). In normal code, you can cast a null reference to any nullable type (whether that's a reference type or a nullable value type). However, if you use the "is" C# operator with a null value, it will always return false. Cast and OfType follow the same rules, basically.

It's worth noting that (as of .NET 3.5 SP1) Cast and OfType only perform reference and unboxing conversions. They *won't* convert a boxed int to a long, or execute user-defined conversions. Basically they follow the same rules as converting from object to a generic type parameter. (That's very convenient for the implementation!) In the original implementation of .NET 3.5, I believe some other conversions *were* supported (in particular, I believe that the boxed int to long conversion *would* have worked). I haven't even attempted to replicate the pre-SP1 behaviour. You can read more details in Ed Maurer's blog post from 2008.

There's one final aspect to discuss: optimization. If "source" already implements IEnumerable<TResult>, the Cast operator just returns the parameter directly, within the original method call. (In other words, this behaviour isn't deferred.) Basically we know that every cast will succeed, so there's no harm in returning the input sequence. This means you *shouldn't* use Cast as an "isolation" call to protect your original data source, in the same way as we sometimes use Select with an identity projection. See Eric Lippert's blog post on degenerate queries for more about protecting the original source of a query.

In the LINQ to Objects implementation, OfType *never* returns the source directly. It *always* uses an iterator. Most of the time, it's probably right to do so. Just because something implements IEnumerable<string> doesn't mean everything within it should be returned by OfType... because some elements may be null. The same is true of an IEnumerable<int?> - but *not* an IEnumerable<int>. For a non-nullable value type T, if source implements IEnumerable<T> then source.OfType<T>() will always contain the exact same sequence of elements as source. It does no more harm to return source from OfType() here than it does from Cast().

What are we going to test?

There are "obvious" tests for deferred execution and eager argument validation. Beyond that, I effectively have two types of test: ones which focus on whether the call returns the original argument, and ones which test the behaviour of iterating over the results (including whether or not an exception is thrown).

The iteration tests are generally not that interesting - in particular, they're similar to tests we've got everywhere else. The "identity" tests are more interesting, because they show some differences between conversions that are allowed by the CLR and those allowed by C#. It's obvious that an array of strings is going to be convertible to IEnumerable<string>, but a test like this might give you more pause for thought:

```
[Test]
public void OriginalSourceReturnedForInt32ArrayToUInt32SequenceConversion()
{
    IEnumerable enums = new int[10];
    Assert.AreSame(enums, enums.Cast<uint>());
}
```

That's trying to "cast" an int[] to an IEnumerable<uint>. If you try the same in normal C# code, it will fail - although if you cast it to "object" first (to distract the compiler, as it were) it's fine at both compile time and execution time:

```
int[] ints = new int[10];
// Fails with CS0030
IEnumerable<uint> uints = (IEnumerable<uint>) ints;

// Succeeds at execution time
IEnumerable<uint> uints = (IEnumerable<uint>)(object) ints;
```

We can have a bit more fun at the compiler's expense, and note its arrogance:

```
int[] ints = new int[10];

if (ints is IEnumerable<uint>)
{
    Console.WriteLine("This won't be printed");
}
if (((object) ints) is IEnumerable<uint>)
{
    Console.WriteLine("This will be printed");
}
```

This generates a warning for the first block "The given expression is never of the provided (...) type" and the compiler has the cheek to remove the block entirely... despite the fact that it *would* have worked if only it had been emitted as code.

Now, I'm not *really* trying to have a dig at the C# team here - the compiler is actually acting entirely reasonably within the rules of C#. It's just that the CLR has subtly different rules around conversions - so when the compiler makes a prediction about

what *would* happen with a particular cast or "is" test, it can be wrong. I don't think this has ever bitten me as an issue, but it's quite fun to watch. As well as this signed/unsigned difference, there are similar conversions between arrays of enums and their underlying types.

There's another type of conversion which is interesting:

```
[Test]
public void OriginalSourceReturnedDueToGenericCovariance()
{
    IEnumerable strings = new List<string>();
    Assert.AreSame(strings, strings.Cast<object>());
}
```

This takes advantage of the generic variance introduced in .NET 4 - sort of. There is now a reference conversion from List<string> to IEnumerable<object> which wouldn't have worked in .NET 3.5. However, this isn't due to the fact that C# 4 now knows about variance; the compiler isn't verifying the conversion here, after all. It isn't due to a new feature in the CLRv4 - generic variance for interfaces and delegates has been present since generics were introduced in CLRv2. It's only due to the change in the IEnumerable<T> type, which has become IEnumerable<out T> in .NET 4. If you could make the same change to the standard library used in .NET 3.5, I believe the test above would pass. (It's possible that the precise CLR rules for variance changed between CLRv2 and CLRv4 - I don't think this variance was widely used before .NET 4, so the risk of it being a problematically-breaking change would have been slim.)

In addition to all these functional tests, I've included a couple of tests to show that the compiler uses Cast in query expressions if you give a range variable an explicit type. This works for both "from" and "join":

```
[Test]
public void CastWithFrom()
{
    IEnumerable strings = new[] { "first", "second", "third" };
    var query = from string x in strings
                select x;
    query.AssertSequenceEqual("first", "second", "third");
}

[Test]
public void CastWithJoin()
{
    var ints = Enumerable.Range(0, 10);
    IEnumerable strings = new[] { "first", "second", "third" };
    var query = from x in ints
                join string y in strings on x equals y.Length
```

```
                select x + ":" + y;
    query.AssertSequenceEqual("5:first", "5:third", "6:second");
}
```

Note how the compile-time type of "strings" is just IEnumerable in both cases. We couldn't use this in a query expression normally, because LINQ requires generic sequences - but by giving the range variables explicit types, the compiler has inserted a call to Cast which makes the rest of the translation work.

Let's implement them!

The "eager argument validation, deferred sequence reading" mode of Cast and OfType means we'll use the familiar approach of a non-iterator-block public method which finally calls an iterator block if it gets that far. This time, however, the optimization occurs within the public method. Here's Cast, to start with:

```
public static IEnumerable<TResult> Cast<TResult>(this IEnumerable source)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    IEnumerable<TResult> existingSequence = source as IEnumerable<TResult>;
    if (existingSequence != null)
    {
        return existingSequence;
    }
    return CastImpl<TResult>(source);
}

private static IEnumerable<TResult> CastImpl<TResult>(IEnumerable source)
{
    foreach (object item in source)
    {
        yield return (TResult) item;
    }
}
```

We're using the normal as/null-test to check whether we can just return the source directly, and in the loop we're casting. We *could* have made the iterator block very slightly shorter here, using the behaviour of foreach to our advantage:

```
foreach (TResult item in source)
{
    yield return item;
}
```

Yikes! Where's the cast gone? How can this possibly work? Well, the cast is still there - it's just been inserted automatically by the compiler. It's the invisible cast that was present in almost every foreach loop in C# 1. The fact that it *is* invisible is the reason I've chosen the previous version. The point of the method is to cast each element - so it's pretty important to make the cast as obvious as possible.

So that's Cast. Now for OfType. First let's look at the public entry point:

```
public static IEnumerable<TResult> OfType<TResult>(this IEnumerable source)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    if (default(TResult) != null)
    {
        IEnumerable<TResult> existingSequence = source as
IEnumerable<TResult>;
        if (existingSequence != null)
        {
            return existingSequence;
        }
    }
    return OfTypeImpl<TResult>(source);
}
```

This is *almost* the same as Cast, but with the additional test of "default(TResult) != null" before we check whether the input sequence is an IEnumerable<TResult>. That's a simple way of saying, "Is this a non-nullablle value type." I don't know for sure, but I'd *hope* that when the JIT compiler looks at this method, it can completely wipe out the test, either removing the body of the if statement completely for nullable value types and reference types, or just go execute the body unconditionally for non-nullable value types. It really doesn't matter if JIT *doesn't* do this, but one day I may get up the courage to tackle this with cordbg and find out for sure... but not tonight.

Once we've decided we've got to iterate over the results ourselves, the iterator block method is quite simple:

```
private static IEnumerable<TResult> OfTypeImpl<TResult>(IEnumerable source)
{
    foreach (object item in source)
    {
        if (item is TResult)
        {
            yield return (TResult) item;
        }
```

```
        }
    }
```

Note that we can't use the "as and check for null" test here, because we don't know that TResult is a nullable type. I was *tempted* to try to write two versions of this code - one for reference types and one for value types. (I've found before that using "as and check for null" is really slow for nullable value types. That may change, of course.) However, that would be quite tricky and I'm not convinced it would have much impact. I did a quick test yesterday testing whether an "object" was actually a "string", and the is+cast approach seemed just as good. I suspect that may be because string is a sealed class, however... testing for an interface or a non-sealed class *may* be more expensive. Either way, it would be premature to write a complicated optimization without testing first.

Conclusion

It's not clear to me why Microsoft optimizes Cast but not OfType. There's a possibility that I've missed a reason why OfType shouldn't be optimized even for a sequence of non-nullable value type values - if you can think of one, please point it out in the comments. My immediate objection would be that it "reveals" the source of the query... but as we've seen, Cast already does that sometimes, so I don't think that theory holds.

Other than that decision, the rest of the *implementation* of these operators has been pretty plain sailing. It did give us a quick glimpse into the difference between the conversions that the CLR allows and the ones that the C# specification allows though, and that's always fun.

Next up - SequenceEqual.

Back to the table of contents.

Part 34 - SequenceEqual

Nearly there now...

What is it?

SequenceEqual has two overloads - the obvious two given that we're dealing with equality:

```
public static bool SequenceEqual<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second)

public static bool SequenceEqual<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second,
    IEqualityComparer<TSource> comparer)
```

The purpose of the operator is to determine if two sequences are equal; that is, if they consist of the same elements, in the same order. A custom equality comparer can be used to compare each individual pair of elements. Characteristics:

- The first and second parameters mustn't be null, and are validated immediately.
- The comparer parameter can be null, in which case the default equality comparer for TSource is used.
- The overload without a comparer uses the default equality comparer for TSource (no funny discrepancies if ICollection<T> instances are involved, this time).
- It uses immediate execution.
- It returns as soon as it notices a difference, without evaluating the rest of either sequence.

So far, so good. Note that no optimizations are mentioned above. There are questions you might consider:

- Should foo.SequenceEqual(foo) always return true?
- If either or both of the sequences implements another collection interface, does that help us?

The first question sounds like it should be a no-brainer, but it's not as simple as it sounds. Suppose we have a sequence which always generates 10 random numbers.

Is it equal to itself? If you iterate over it twice, you'll usually get different results. What about a sequence which explicitly changes each time you iterate over it, based on some side-effect? Both the .NET and Edulinq implementations say that these are non-equal. (The random case is interesting, of course - it *could* happen to yield the same elements as we iterate over the two sequences.)

The second question feels a little simpler to me. We can't take a shortcut to returning true, but it seems reasonably obvious to me that if you have two collections which allow for a fast "count" operation, and the two counts are different, then the sequences are unequal. Unfortunately, LINQ to Objects appears *not* to optimize for this case: if you create two huge arrays of differing sizes but equal elements as far as possible, it will take a long time for SequenceEqual to return false. Edulinq *does* perform this optimization. Note that just having *one* count isn't useful: you might expect it to, but it turns out that by the time we could realize that the lengths were different in that case, we're about to find that out in the "normal" fashion anyway, so there's no point in complicating the code to make use of the information.

What are we going to test?

As well as the obvious argument validation, I have tests for:

- Collections of different lengths
- Ranges of different lengths, both with first shorter than second and vice versa
- Using a null comparer
- Using a custom comparer
- Using no comparer
- Equal arrays
- Equal ranges
- The non-optimization of foo.SequenceEquals(foo) (using side-effects)
- The optimization using Count (fails on LINQ to Objects)
- Ordering: { 1, 2 } should not be equal to { 2, 1 }
- The use of a HashSet<string> with a case-insensitive comparer: the default (case-sensitive) comparer is still used when no comparer is provided
- Infinite first sequence with finite second sequence, and vice versa
- Sequences which differ just before they would go bang

None of the test code is particularly interesting, to be honest.

Let's implement it!

I'm not going to show the comparer-less overoad, as it just delegates to the one with a comparer.

Before we get into the guts of SequenceEqual, it's time for a bit of refactoring. If we're going to optimize for count, we'll need to perform the same type tests as Count() twice. That would be horrifically ugly inline, so let's extract the functionality out into a private method (which Count() can then call as well):

```
private static bool TryFastCount<TSource>(
    IEnumerable<TSource> source,
    out int count)
{
    // Optimization for ICollection<T>
    ICollection<TSource> genericCollection = source as ICollection<TSource>;
    if (genericCollection != null)
    {
        count = genericCollection.Count;
        return true;
    }

    // Optimization for ICollection
    ICollection nonGenericCollection = source as ICollection;
    if (nonGenericCollection != null)
    {
        count = nonGenericCollection.Count;
        return true;
    }
    // Can't retrieve the count quickly. Oh well.
    count = 0;
    return false;
}
```

Pretty simple. Note that we *always* have to set the out parameter to some value. We use 0 on failure - which happens to work out nicely in the Count, as we can just start incrementing if TryFastCount has returned false.

Now we can make a start on SequenceEqual. Here's the skeleton before we start doing the real work:

```
public static bool SequenceEqual<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second,
    IEqualityComparer<TSource> comparer)
{
    if (first == null)
    {
        throw new ArgumentNullException("first");
    }
    if (second == null)
    {
        throw new ArgumentNullException("second");
    }
```

```
    int count1;
    int count2;
    if (TryFastCount(first, out count1) && TryFastCount(second, out count2))
    {
        if (count1 != count2)
        {
            return false;
        }
    }

    comparer = comparer ?? EqualityComparer<TSource>.Default;

    // Main part of implementation goes here
}
```

I *could* have included the comparison between count1 and count2 within the single "if" condition, like this:

```
if (TryFastCount(first, out count1) &&
    TryFastCount(second, out count2) &&
    count1 != count2)
{
    return false;
}
```

... but I don't usually like using the values of out parameters like this. The behaviour is well-defined and correct, but it just feels a little ugly to me.

Okay, now let's implement the "Main part" which at the bottom of the skeleton. The idea is simple:

- Get the iterators for both sequences
- Use the iterators "in parallel" (not in the multithreading sense, but in the movement of the logical cursor down the sequence) to compare pairs of elements; we can return false if we ever see an unequal pair
- If we ever see that one sequence has finished and the other hasn't, we can return false
- If we get to the end of both sequences in the same iteration, we can return true

I've got three different ways of representing the basic algorithm in code though. Fundamentally, the problem is that we don't have a way of iterating over pairs of elements in two sequences with foreach - we can't use one foreach loop inside another, for hopefully obvious reasons. So we'll have to call GetEnumerator() explicitly on *at least one* of the sequences... and we could do it for both if we want.

The first implementation (and my least favourite) *does* use a foreach loop:

```
using (IEnumerator<TSource> iterator2 = second.GetEnumerator())
{
    foreach (TSource item1 in first)
    {
        // second is shorter than first
        if (!iterator2.MoveNext())
        {
            return false;
        }
        if (!comparer.Equals(item1, iterator2.Current))
        {
            return false;
        }
    }
    // If we can get to the next element, first was shorter than second.
    // Otherwise, the sequences are equal.
    return !iterator2.MoveNext();
}
```

I don't have a desperately good reason for picking them this way round (i.e. foreach over first, and GetEnumerator() on second) other than that it seems to still give primacy to first somehow... only first gets the "special treatment" of a foreach loop. (I can almost hear the chants now, "Equal rights for second sequences! Don't leave us out of the loop! Stop just 'using' us!") Although I'm being frivolous, I dislike the asymmetry of this.

The second attempt is a half-way house: it's still asymmetric, but slightly less so as we're explicitly fetching both iterators:

```
using (IEnumerator<TSource> iterator1 = first.GetEnumerator(),
                            iterator2 = second.GetEnumerator())
{
    while (iterator1.MoveNext())
    {
        // second is shorter than first
        if (!iterator2.MoveNext())
        {
            return false;
        }
        if (!comparer.Equals(iterator1.Current, iterator2.Current))
        {
            return false;
        }
    }
    // If we can get to the next element, first was shorter than second.
    // Otherwise, the sequences are equal.
    return !iterator2.MoveNext();
```

```
}
```

Note the use of the multi-variable "using" statement; this is equivalent nesting one statement inside another, of course.

The similarities between these two implementations are obvious - but the differences are worth pointing out. In the latter approach, we call MoveNext() on both sequences, and then we access the Current property on both sequences. In each case we use iterator1 before iterator2, but it still feels like they're being treated more equally somehow. There's still the fact that iterator1 is being used in the while loop condition, whereas iterator2 has to be used both inside and outside the while loop. Hmm.

The third implementation takes this even further, changing the condition of the while loop:

```
using (IEnumerator<TSource> iterator1 = first.GetEnumerator(),
       iterator2 = second.GetEnumerator())
{
    while (true)
    {
        bool next1 = iterator1.MoveNext();
        bool next2 = iterator2.MoveNext();
        // Sequences aren't of same length. We don't
        // care which way round.
        if (next1 != next2)
        {
            return false;
        }
        // Both sequences have finished - done
        if (!next1)
        {
            return true;
        }
        if (!comparer.Equals(iterator1.Current, iterator2.Current))
        {
            return false;
        }
    }
```

This feels about as symmetric as we can get. The use of next1 in the middle "if" condition is incidental - it could just as easily be next2, as we know the values are equal. We could switch round the order of the calls to MoveNext(), the order of arguments to comparer.Equals - the *structure* is symmetric.

I'm not generally a fan of while(true) loops, but in this case I think I rather like it. It makes it obvious that we're going to keep going until we've got a good reason to stop: one of the three return statements. (I suppose I should apologise to fans of the

dogma around single exit points for methods, if any are reading. This must be hell for you...)

Arguably this is all a big fuss about nothing - but writing Edulinq has given me a new appreciation for diving into this level of detail to find the most readable code. As ever, I'd be interested to hear your views. (All three versions are in source control. Which one is active is defined with a #define.)

Conclusion

I really don't know why Microsoft didn't implement the optimization around different lengths for SequenceEqual. Arguably in the context of LINQ you're *unlikely* to be dealing with two materialized collections at a time - it's much more common to have one collection and a lazily-evaluated query, or possibly just two queries... but it's a cheap optimization and the benefits can be significant. *Maybe* it was just an oversight.

Our next operator also deals with pairs of elements, so we may be facing similar readability questions around it. It's Zip - the only new LINQ query operator in .NET 4.

Back to the table of contents.

Part 35 - Zip

Zip will be a familiar operator to any readers who use Python. It was introduced in .NET 4 - it's not entirely clear why it wasn't part of the first release of LINQ, to be honest. Perhaps no-one thought of it as a useful operator until it was too late in the release cycle, or perhaps implementing it in the other providers (e.g. LINQ to SQL) took too long. Eric Lippert blogged about it in 2009, and I find it interesting to note that aside from braces, layout and names we've got exactly the same code. (I read the post at the time of course, but implemented it tonight without looking back at what Eric had done.) It's not exactly surprising, given how trivial the implementation is. Anyway, enough chit-chat...

What is it?

Zip has a single signature, which isn't terribly complicated:

```
public static IEnumerable<TResult> Zip<TFirst, TSecond, TResult>(
    this IEnumerable<TFirst> first,
    IEnumerable<TSecond> second,
    Func<TFirst, TSecond, TResult> resultSelector)
```

Just from the signature, the name, and experience from the rest of this blog series it should be easy enough to guess what Zip does:

- It uses deferred execution, not reading from either sequence until the result sequence is read
- All three parameters must be non-null; this is validated eagerly
- Both sequences are iterated over "at the same time": it calls GetEnumerator() on each sequence, then moves each iterator forward, then reads from it, and repeats.
- The result selector is applied to each pair of items obtained in this way, and the result yielded
- It stops when *either* sequence terminates
- As a natural consequence of how the sequences are read, we don't need to perform any buffering: we only care about one element from each sequence at a time.

There are really only two things that I could see might have been designed differently:

- It could have just returned IEnumerable<Tuple<TFirst, TSecond>> but that would have been less efficient in many cases (in terms of the GC) and

inconsistent with the rest of LINQ
- It could have provided different options for what to do with sequences of differents lengths. For example:
  - Throw an exception
  - Use the default value of the shorter sequence type against the remaining items of the longer sequence
  - Use a *specified* default value of the shorter sequence in the same way

I don't have any problem with the design that's been chosen here though.

What are we going to test?

There are no really interesting test cases here. We test argument validation, deferred execution, and the obvious "normal" cases. I do have tests where "first" is longer than "second" and vice versa.

The one test case which is noteworthy isn't really present for the sake of testing at all - it's to demonstrate a technique which can occasionally be handy. Sometimes we really want to perform a projection on adjacent pairs of elements. Unfortunately there's no LINQ operator to do this naturally (although it's easy to write one) but Zip can provide a workaround, so long as we don't mind evaluating the sequence twice. (That could be a problem in some cases, but is fine in others.)

Obviously if you just zip a sequence with itself directly you get each element paired with the same one. We effectively need to "shift" or "delay" one sequence somehow. We can do this using Skip, as shown in this test:

```
[Test]
public void AdjacentElements()
{
    string[] elements = { "a", "b", "c", "d", "e" };
    var query = elements.Zip(elements.Skip(1), (x, y) => x + y);
    query.AssertSequenceEqual("ab", "bc", "cd", "de");
}
```

It always takes me a little while to work out whether I want to make first skip or second - but if we want the second element as the first element of second (try getting that right ten times in a row - it makes sense, honest!) means that we want to call Skip on the sequence used as the argument for second. Obviously it would work the other way round too - we'd just get the pairs presented with the values switched, so the results of the query above would be "ba", "cb" etc.

Let's implement it!

Guess what? It's yet another operator with a split implementation between the argument validation and the "real work". I'll skip argument validation, and get into the tricky stuff. Are you ready? Sure you don't want another coffee?

```
private static IEnumerable<TResult> ZipImpl<TFirst, TSecond, TResult>(
    IEnumerable<TFirst> first,
    IEnumerable<TSecond> second,
    Func<TFirst, TSecond, TResult> resultSelector)
{
    using (IEnumerator<TFirst> iterator1 = first.GetEnumerator())
    using (IEnumerator<TSecond> iterator2 = second.GetEnumerator())
    {
        while (iterator1.MoveNext() && iterator2.MoveNext())
        {
            yield return resultSelector(iterator1.Current,
iterator2.Current);
        }
    }
}
```

Okay, so possibly "tricky stuff" was a bit of an overstatement. Just about the only things to note are:

- I've "stacked" the using statements instead of putting the inner one in braces and indenting it. For using statements with different variable types, this is one way to keep things readable, although it can be a pain when tools try to reformat the code. (Also, I don't *usually* omit optional braces like this. It does make me feel a bit dirty.)
- I've used the "symmetric" approach again instead of a using statement with a foreach loop inside it. That wouldn't be *hard* to do, but it wouldn't be as simple.

That's just about it. The code does exactly what it looks like, which doesn't make for a very interesting blog post, but does make for good readability.

Conclusion

Two operators to go, one of which I might not even tackle fully (AsQueryable - it *is* part of Queryable rather than Enumerable, after all).

AsEnumerable should be pretty easy...

Back to the table of contents.

Part 36 - AsEnumerable

Our last operator is the simplest of all. Really, *really* simple.

What is it?

AsEnumerable has a single signature:

```
public static IEnumerable<TSource> AsEnumerable<TSource>(this
IEnumerable<TSource> source)
```

I can describe its behaviour pretty easily: it returns source.

That's all it does. There's no argument validation, it doesn't create another iterator. It just returns source.

You may well be wondering what the point is... and it's all about changing the compile-time type of the expression. I'm going to take about IQueryable<T> in another post (although probably not implement anything related to it) but hopefully you're aware that it's usually used for "out of process" queries - most commonly in databases.

Now it's not entirely uncommon to want to perform some aspects of the query in the database, and then a bit more manipulation in .NET - particularly if there are aspects you basically can't implement in LINQ to SQL (or whatever provider you're using). For example, you may want to build a particular in-memory representation which isn't really amenable to the provider's model.

In that case, a query can look something like this:

```
var query = db.Context
              .Customers
              .Where(c => some filter for SQL)
              .OrderBy(c => some ordering for SQL)
              .Select(c => some projection for SQL)
              .AsEnumerable() // Switch to "in-process" for rest of query
              .Where(c => some extra LINQ to Objects filtering)
              .Select(c => some extra LINQ to Objects projection);
```

All we're doing is changing the compile-time type of the sequence which is propagating through our query from IQueryable<T> to IEnumerable<T> - but that

means that the compiler will use the methods in Enumerable (taking delegates, and executing in LINQ to Objects) instead of the ones in Queryable (taking *expression trees*, and usually executing out-of-process).

Sometimes we *could* do this with a simple cast or variable declaration. However, for one thing that's ugly, whereas the above query is fluent and quite readable, so long as you appreciate the importance of AsEnumerable. The more important point is that it's not always possible, because we may very well be dealing with a sequence of an anonymous type. An extension method lets the compiler use type inference to work out what the T should be for IEnumerable<T>, but you can't actually express that in your code.

In short - it's not nearly as useless an operator as it seems at first sight. That doesn't make it any more complicated to test or implement though...

What are we going to test?

In the spirit of exhaustive testing, I have actually tested:

- A normal sequence
- A null reference
- A sequence which would throw an exception if you actually tried to use it

The tests just assert that the result is the same reference as we've passed in.

I have one additional test which comes as close as I can to demonstrating the point of AsEnumerable without using Queryable:

```
[Test]
public void AnonymousType()
{
    var list = new[] {
        new { FirstName = "Jon", Surname = "Skeet" },
        new { FirstName = "Holly", Surname = "Skeet" }
    }.ToList();

    // We can't cast to IEnumerable<T> as we can't express T.
    var sequence = list.AsEnumerable();
    // This will now use Enumerable.Contains instead of List.Contains
    Assert.IsFalse(sequence.Contains(new { FirstName = "Tom", Surname =
"Skeet" }));
}
```

And finally...

Let's implement it!

There's not much scope for an interesting implementation here I'm afraid. Here it is, in its totality:

```
public static IEnumerable<TSource> AsEnumerable<TSource>(this
IEnumerable<TSource> source)
{
    return source;
}
```

It feels like a fittingly simple end to the Edulinq implementation.

Conclusion

I *think* that's all I'm going to actually implement from LINQ to Objects. Unless I've missed something, that covers all the methods of Enumerable from .NET 4.

That's not the end of this series though. I'm going to take a few days to write up some thoughts about design choices, optimizations, other operators which might have been worth including, and a little bit about how IQueryable<T> works.

Don't forget that the source code is freely available on Google Code. I'll be happy to patch any embarrassing bugs :)

Back to the table of contents.

Part 37 - Guiding principles

Now that I'm "done" reimplementing LINQ to Objects - in that I've implemented all the methods in System.Linq.Enumerable - I wanted to write a few posts looking at the bigger picture. I'm not 100% sure of what this will consist of yet; I want to avoid this blog series continuing forever. However, I'm confident it will contain (in no particular order):

- This post: principles governing the behaviour of LINQ to Objects
- Missing operators: what else I'd have liked to see in Enumerable
- Optimization: where the .NET implementation could be further optimized, and why some obvious-sounding optimizations may be inappropriate
- How query expression translations work, in brief (and with a cheat sheet)
- The difference between IQueryable<T> and IEnumerable<T>
- Sequence identity, the "Contains" issue, and other knotty design questions
- Running the Edulinq tests against other implementations

If there are other areas you want me to cover, please let me know.

The principles behind the LINQ to Objects implementation

The design LINQ to Objects is built on a few guiding principles, both in terms of design and implementation details. You need to understand these, but also implementations should be clear about what they're doing in these terms too.

Extension method targets and argument validation

IEnumerable<T> is the core sequence type, not just for LINQ but for .NET as a whole. Almost *everything* is written in terms of IEnumerable<T> at least as input, with the following exceptions:

- Empty, Range and Repeat don't have input sequences (these are the only non-extension methods)
- OfType and Cast work on the non-generic IEnumerable type instead
- ThenBy and ThenByDescending work on IOrderedEnumerable<T>

All operators other than AsEnumerable verify that any input sequence is non-null. This validation is performed eagerly (i.e. when the method is called) even if the operator uses deferred execution for the results. Any delegate used (typically a projection or predicate of some kind) must be non-null. Again, this validation is performed eagerly.

IEqualityComparer<T> is used for all custom equality comparisons. Any parameter of this type *may* be null, in which case the default equality comparer for the type is used. In *most* cases the default equality comparer for the type is also used when no custom equality comparer is used, but Contains has some odd behaviour around this. Equality comparers are expected to be able to handle null values. IComparer<T> is *only* used by the OrderBy/ThenBy operators and their descending counterparts - and only then if you want custom comparisons between keys. Again, a null IComparer<T> means "use the default for the type"

Timing of input sequence "opening"

Any operator with a return type of IEnumerable<T> or IOrderedEnumerable<T> uses *deferred execution*. This means that the method doesn't read anything from any input sequences until someone starts reading from the result sequence. It's not clearly defined exactly *when* input sequences will first be accessed - for some operators if may be when GetEnumerator() is called; for others it may be on the first call to MoveNext() on the resulting iterator. Callers should not depend on these slight variations. Deferred execution is common for operators in the middle of queries. Operators which use deferred execution effectively represent queries rather than the results of queries - so if you change the contents of the original source of the query and then iterate over the query itself again, you'll see the change. For example:

```
List<string> source = new List<string>();
var query = source.Select(x => x.ToUpper());

// This loop won't write anything out
foreach (string x in query)
{
    Console.WriteLine(x);
}

source.Add("foo");
source.Add("bar");

// This loop will write out "FOO" and "BAR" - even
// though we haven't changed the value of "query"
foreach (string x in query)
{
    Console.WriteLine(x);
}
```

Deferred execution is one of the hardest parts of LINQ to understand, but once you do, everything becomes somewhat simpler.

All other operators use *immediate execution*, fetching all the data they need from the

input before they return a value... so that by the time they *do* return, they will no longer see or care about changes to the input sequence. For operators returning a scalar value (such as Sum and Average) this is blatantly obvious - the value of a variable of type double isn't going to change just because you've added something to a list. However, it's slightly less for the "ToXXX" methods: ToLookup, ToArray, ToList and ToDictionary. These do *not* return views on the original sequence, unlike the "As" methods: AsEnumerable which we've seen, and Queryable.AsQueryable which I didn't implement. Focus on the prefix part of the name: the "To" part indicates a conversion to a particular type. The "As" prefix indicates a wrapper of some kind. This is consistent with other parts of the framework, such as List<T>.AsReadOnly and Array.AsReadOnly<T>.

Very importantly, LINQ to Objects only iterates over any input sequence at most **once**, whether the execution is deferred or immediate. Some operators would be easier to implement if you could iterate over the input twice - but it's important that they don't do so. Of course if you provide the same sequence for two inputs, it will treat those as logically different sequences. Similarly if you iterate over a result sequence more than once (for operators that return IEnumerable<T> or a related interface, rather than List<T> or an array etc), that will iterate over the input sequence again.

This means it's fine to use LINQ to Objects with sequences which may only be read once (such as a network stream), or which are relatively expensive to reread (imagine a log file reader over a huge set of logs) or which give inconsistent results (imagine a sequence of random numbers). In some cases it's okay to use LINQ to Objects with an infinite sequence - in others it's not. It's *usually* fairly obvious which is the case.

Timing of input sequence reading, and memory usage

Where possible within deferred execution, operators act in a *streaming* fashion, only reading from the input sequence when they have to, and "forgetting" data as soon as they can. This allows for long - potentially infinite - sequences to be handled elegantly without memory running out.

Some operators naturally need to read all the data in before they can return anything. The most obvious example of this is Reverse, which will always yield the last element of the input stream as the first element in the result stream.

A third pattern occurs with operators such as Distinct, which yield data as they go, but accumulate elements too, taking more and more memory until the caller stops iterating (usually either by jumping out of the foreach loop, or letting it terminate naturally).

Where an operator takes two input sequences - such as Join - you need to understand the consumption of each one separately. For example, Join uses deferred execution, but as soon as you ask for the first element of the result set, it will read the "second" sequence *completely* and buffer it - whereas the "first" sequence is streamed. This isn't the case for all operators with two inputs, of course - Zip streams both input sequences, for example. Check the documentation - and the relevant Edulinq blog post - for details.

Obviously any operator which uses immediate execution has to read all the data it's interested in before it returns. This doesn't necessarily mean they will read to the end of the sequence though, and they may not need to buffer the data they read. (Simple examples are ToList which has to keep everything, and Sum which doesn't.)

Queries vs data

Closely related to the details of when the input is read is the concept of what the result of an operator actually represents. Operators which use deferred execution return *queries*: each time you iterate over the result sequence, the query will look at the input sequence again. The query itself doesn't contain the data - it just knows how to get at the data.

Operators which use immediate execution work the other way round: they read all the data they need, and then forget about the input sequence. For operators like Average and Sum this is obvious as it's just a simple scalar value - but for operators like ToList, ToDictionary, ToLookup and ToArray, it means that the operator has to make a copy of everything it needs. (This is potentially a *shallow* copy of course - depending on what user-defined projections are applied. The normal behaviour of mutable reference types is still valid.)

I realise that in many ways I've just said the same thing multiple times now - but hopefully that will help this crucial aspect of LINQ behaviour sink in, if you were still in any doubt.

Exception handling

I'm unaware of any situation in which LINQ to Objects will catch an exception. If your predicate or projection throws an exception, it will propagate in the obvious way.

However, LINQ to Objects *does* ensure that any iterator it reads from is disposed appropriately - assuming that the caller disposes of any result sequences properly, of course. Note that the foreach statement implicitly disposes of the iterator in a finally block.

Optimization

Various operators are optimized when they detect at execution time that the input sequence they're working on offers a shortcut.

The types most commonly detected are:

- ICollection<T> and ICollection for their Count property
- IList<T> for its random access indexer

I'll look at optimization in much more detail in a separate post.

Conclusion

This post has not been around the guiding principles behind LINQ itself - lambda calculus or anything like that. It's more been a summary of the various aspects of behaviour we've seen across the various operators we've implemented. They're the rules I've had to follow in order to make Edulinq reasonably consistent with LINQ to Objects.

Next time I'll talk about some of the operators which I think *should* have made it into the core framework, at least for LINQ to Objects.

Back to the table of contents.

Part 38 - What's missing?

I mentioned before that the Zip operator was only introduced in .NET 4, so clearly there's a *little* wiggle room for LINQ to Object's query operators to grow in number. This post mentions some of the ones I think are most sorely lack - either because I've wanted them myself, or because I've seen folks on Stack Overflow want them for entirely reasonable use cases.

There is an issue with respect to other LINQ providers, of course: as soon as some useful operators are available for LINQ to Objects, there will be people who want to apply them to LINQ to SQL, the Entity Framework and the like. Worse, if they're *not* included in Queryable with overloads based on expression trees, the LINQ to Objects implementation will silently get picked - leading to what looks like a lovely query performing like treacle while the client slurps over the entire database. If they *are* included in Queryable, then third party LINQ providers could end up with a nasty versioning problem. In other words, some care is needed and I'm glad I'm not the one who has to decide how new features are introduced.

I've deliberately *not* looked at the extra set of operators introduced in the System.Interactive part of Reactive Extensions... nor have I looked back over what we've implemented in MoreLINQ (an open source project I started specifically to create new operators). I figured it would be worth thinking about this afresh - but look at both of those projects for actual implementations instead of just ideas.

Currently there's no implementation of any of this in Edulinq - but I could potentially create an "Edulinq.Extras" assembly which made it all available. Let me know if any of these sounds particularly interesting to see in terms of implementation.

FooBy

I *love* OrderBy and ThenBy, with their descending cousins. They're so much cleaner than building a custom comparer which just performs a comparison between two properties. So why stop with ordering? There's a whole bunch of operators which could do with some "FooBy" love. For example, imagine we have a list of files, and we want to find the longest one. We don't want to perform a total ordering by size descending, nor do we want to find the maximum file size itself: we want the file *with* the maximum size. I'd like to be able to write that query as:

```
FileInfo biggestFile = files.MaxBy(file => file.Length);
```

Note that we can get a *similar* result by performing one pass to find the maximum length, and then another pass to find the file with that length. However, that's inefficient and assumes we can read the sequence twice (and get the same results both times). There's no need for that. We could get the same result using Aggregate with a pretty complicated aggregation, but I think this is a sufficiently common case to deserve its own operator.

We'd want to specify which value would be returned if multiple files had the same length (my suggestion would be the first one we encountered with that length) and we could also specify a key comparer to use. The signatures would look like this:

```
public static TSource MaxBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector)

public static TSource MaxBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IComparer<TKey> comparer)
```

Now it's not just Max and Min that gain from this "By" idea. It would be useful to apply the same idea to the set operators. The simplest of these to think about would be DistinctBy, but UnionBy, IntersectBy and ExceptBy would be reasonable too. In the case of ExceptBy and IntersectBy we *could* potentially take the key collection to indicate the keys of the elements we wanted to exclude/include, but it would probably be more consistent to force the two input sequences to be of the same type (as they would have to be for UnionBy and IntersectBy of course). ContainsBy *might* be useful, but that would effectively be a Select followed by a normal Contains - possibly not useful enough to merit its own operator.

TopBy and TopByDescending

These may sound like they belong in the FooBy section, but they're somewhat different: they're effectively specializations of OrderBy and OrderByDescending where you already know how many elements you want to preserve. The return type would be IOrderedEnumerable<T> so you could still use ThenBy/ThenByDescending as normal. That would make the following two queries equivalent - but the second *might* be a lot more efficient than the first:

```
var takeQuery = people.OrderBy(p => p.LastName)
                      .ThenBy(p => p.FirstName)
                      .Take(3);

var topQuery = people.TopBy(p => p.LastName, 3)
```

```
                    .ThenBy(p => p.FirstName);
```

An implementation could easily delegate to various different strategies depending on the number given - for example, if you asked for more than 10 values, it may not be worth doing anything more than a simple sort and restrict the output. If you asked for just the top 3 values, that could return an IOrderedEnumerable implementation specifically hard-coded to 3 values, etc.

Aside from anything else, if you were confident in what the implementation did (and that's a *very* big "if") you could use a potentially huge input sequence with such a query - larger than you could fit into memory in one go. That's fine if you're only keeping the top three values you've seen so far, but would fail for a complete ordering, even one which was able to yield results before performing *all* the ordering: if it doesn't *know* you're going to stop after three elements, it can't throw anything away.

Perhaps this is too specialized an operator - but it's an interesting one to think about. It's worth noting that this probably only makes sense for LINQ to Objects, which never gets to see the whole query in one go. Providers like LINQ to SQL can optimize queries of the form OrderBy(...).ThenBy(...).Take(...) because by the time they need to translate the query into SQL, they will have an expression tree representation which includes the "Take" part.

TryFastCount and TryFastElementAt

One of the implementation details of Edulinq is its TryFastCount method, which basically encapsulates the logic around attempting to find the count of a sequence if it implements ICollection or ICollection<T>. Various built-in LINQ operators find this useful, and anyone writing their own operators has a reasonable chance of bumping into it as well. It seems pointless to duplicate the code all over the place... why not expose it? The signatures might look something like this:

```
public static bool TryFastCount<TSource>(
    this IEnumerable<TSource> source,
    out int count)

public static bool TryFastElementAt<TSource>(
    this IEnumerable<TSource> source,
    int index,
    out TSource value)
```

I would expect TryFastElementAt to use the indexer if the sequence implemented IList<T> without performing any validation: that ought to be the responsibility of the

caller. TryFastCount could use a Nullable<int> return type instead of the return value / out parameter split, but I've kept it consistent with the methods which exist elsewhere in the framework

Scan and SelectAdjacent

These are related operators in that they deal with wanting a more global view than just the current element. Scan would act similarly to Aggregate - except that it would yield the accumulator value after each element. Here's an example of keeping a running total:

```
// Signature:
public static IEnumerable<TAccumulate> Scan<TSource, TAccumulate>(
    this IEnumerable<TSource> source,
    TAccumulate seed,
    Func<TAccumulate, TSource, TAccumulate> func)


int[] source = new int[] { 3, 5, 2, 1, 4 };
var query = source.Scan(0, (current, item) => current + item);
query.AssertSequenceEqual(3, 8, 10, 11, 15);
```

There *could* be a more complicated overload with an extra conversion from TAccumulate to an extra TResult type parameter. That would let us write a Fibonacci sequence query in one line, if we really wanted to...

The SelectAdjacent operator would simply present a selector function with pairs of adjacent items. Here's a similar example, this time calculating the difference between each pair:

```
// Signature:
public static IEnumerable<TResult> SelectAdjacent<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TSource, TResult> selector)


int[] source = new int[] { 3, 5, 2, 1, 4 };
var query = source.SelectAdjacent((current, next) => next - current);
query.AssertSequenceEqual(2, -3, -1, 3);
```

One oddity here is that the result sequence always contains one item fewer than the source sequence. If we wanted to keep the length the same, there are various approaches we could take - but the best one would depend on the situation.

This sounds like a pretty obscure operator, but I've actually seen quite a few LINQ

questions on Stack Overflow where it could have been useful. Is it useful often *enough* to deserve its own operator? Maybe... maybe not.

DelimitWith

This one is really just a bit of a peeve - but again, it's a pretty common requirement. We often want to take a sequence and create a single string which is (say) a comma-delimited version. Yay, String.Join does exactly what we need - particularly in .NET 4, where there's an overload taking IEnumerable<T> so you don't need to convert it to a string array first. However, it's still a *static* method on string - and the name "Join" also looks slightly odd in the context of a LINQ query, as it's got nothing to do with a LINQ-style join.

Compare these two queries: which do you think reads better, and feels more "natural" in LINQ?

```
// Current state of play...
var names = string.Join(",",
                        people.Where(p => p.Age < 18)
                              .Select(p => p.FirstName));

// Using DelimitWith
var names = people.Where(p => p.Age < 18)
                  .Select(p => p.FirstName)
                  .DelimitWith(",");
```

I know which I prefer :)

ToHashSet

(Added on February 23rd 2011.)

I'm surprised I missed this one first time round - I've bemoaned its omission in various places before now. It's easy to create a list, dictionary, lookup or array from an anonymous type, but you can't create a set that way. That's mad, given how simple the relevant operator is, even with an overload for a custom equality comparer:

```
public static HashSet<TSource> ToHashSet<TSource>(
    this IEnumerable<TSource> source)
{
    return source.ToHashSet(EqualityComparer<TSource>.Default);
}

public static HashSet<TSource> ToHashSet<TSource>(
    this IEnumerable<TSource> source,
```

```
    IEqualityComparer<TSource> comparer)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    return new HashSet<TSource>(source, comparer ??
EqualityComparer<TSource>.Default);
}
```

This also makes it much simpler to create a HashSet in a readable way from an existing query expression, without either wrapping the whole query in the constructor call or using a local variable.

Conclusion

These are just the most useful extra methods I thought of, based on the kinds of query folks on Stack Overflow have asked about. I think it's interesting that some are quite general - MaxBy, ExceptBy, Scan and so on - whereas others (TopBy, SelectAdjacent and particularly DelimitWith) are simply aimed at making some very specific but common situations simpler. It feels to me like the more general operators really are *missing* from LINQ - they would fit quite naturally - but the more specific ones probably deserve to be in a separate static class, as "extras".

This is only scratching the surface of what's possible, of course - System.Interactive.EnumerableEx in Reactive Extensions has *loads* of options. Some of them are deliberate parallels of the operators in Observable, but plenty make sense on their own too.

One operator you may have *expected* to see in this list is ForEach. This is a controversial topic, but Eric Lippert has written about it very clearly (no surprise there, then). Fundamentally LINQ is about *querying* a sequence, not taking *action* on it. ForEach breaks that philosophy, which is why I haven't included it here. Usually a foreach statement is a perfectly good alternative, and make the "action" aspect clearer.

Back to the table of contents.

Part 39 - Comparing implementations

While implementing Edulinq, I only focused on two implementations: .NET 4.0 and Edulinq. However, I was aware that there were other implementations available, notably LinqBridge and the one which comes with Mono. Obviously it's interesting to see how other implementations behave, so I've now made a few changes in order to make the test code run in these different environments.

The test environments

I'm using Mono 2.8 (I can't remember the minor version number offhand) but I tend to think of it as "Mono 3.5" or "Mono 4.0" depending on which runtime I'm using and which base libraries I'm compiling against, to correspond with the .NET versions. Both runtimes ship as part of Mono 2.8. I will use these version numbers for this post, and ask forgiveness for my lack of precision: whenever you see "Mono 3.5" please just think "Mono 2.8 running against the 2.0 runtime, possibly using some of the class libraries normally associated with .NET 3.5".

LinqBridge is a bit like Edulinq - a clean room implementation of LINQ to Objects, but built against .NET 2.0. It contains its own Func delegate declarations and its own version of ExtensionAttribute for extension methods. In my experience this makes it difficult to use with the "real" .NET 3.5, so my build targets .NET 2.0 when running against LinqBridge. This means that tests using HashSet had to be disabled. The version of LinqBridge I'm running against is 1.2 - the latest binary available on the web site. This has AsEnumerable as a plain static method rather than an extension method; the code has been fixed in source control, but I wanted to run against a prebuilt binary, so I've just disabled my own AsEnumerable tests for LinqBridge. Likewise the tests for Zip are disabled both for LinqBridge and the "Mono 3.5" tests as Zip was only introduced in .NET 4.

The other issue of not having .NET 4 available in the tests is that the string.Join<T>(string, IEnumerable<T>) overload is unavailable - something I'd used quite a lot in the test code. I've created a new static class called "StringEx" and replaced string.Join with StringEx.Join everywhere.

There are batch files under a new "testing" directory which will build and run:

- Microsoft's LINQ to Objects and Edulinq under .NET
- LinqBridge, Mono 3.5's LINQ to Objects and Edulinq under Mono 3.5
- Mono 4.0's LINQ to Objects and Edulinq under Mono 4.0

Although I have LinqBridge running under .NET 2.0 in Visual Studio, it's a bit of a pain building the tests from a batch file (at least without just calling msbuild). The failures running under Mono 3.5 are the same as those running under .NET 2.0 as far as I can tell, so I'm not too worried.

Note that while I have built the Mono tests under both the 3.5 and 4.0 profiles, the results were the same other than due to generic variance, so I've only included the results of the 4.0 profile below.

What do the tests cover?

Don't forget that the Edulinq tests were written in the spirit of investigation. They cover aspects of LINQ's behaviour which are not guaranteed, both in terms of optimization and simple correctness of behaviour. I have included a test which demonstrates the "issue" with calling Contains on an ICollection<T> which uses a non-default equality comparer, as well as the known issue with OrderByDescending using a comparer which returns int.MinValue. There are optimizations which are present in Edulinq but not in LINQ to Objects, and I have tests for those, too.

The tests which fail against Microsoft's implementation (for known reasons) are normally marked with an [Ignore] attribute to prevent them from alarming me unduly during development. NUnit categories would make more sense here, but I don't believe ReSharper supports them, and that's the way I run the tests normally. Likewise the tests which take a very long time (such as counting more than int.MaxValue elements) are normally suppressed.

In order to truly run *all* my tests, I now have a horrible hack using conditional compilation: if the ALL_TESTS preprocessor symbol is defined, I build my own IgnoreAttribute class in the Edulinq.Tests namespace, which effectively takes precedence over the NUnit one... so NUnit will ignore the [Ignore], so to speak. Frankly all this conditional compilation is pretty horrible, and I wouldn't use it for a "real" project, but this is a slightly unusual situation.

EDIT: It turns out that ReSharper *does* support categories. I'm not sure how far that support goes yet, but at the very least there's "Group by categories" available. I may go through *all* my tests and apply a category to each one: optimization, execution mode, time-consuming etc. We'll see whether I can find the energy for that :)

So, let's have a look at what the test results are...

Edulinq

Unsurprisingly, Edulinq passes all its own tests, with the minor exception of

CastTest.OriginalSourceReturnedDueToGenericCovariance running under Mono 3.5, which doesn't include covariance. Arguably this test should be conditionalised to not even run in that situation, as it's not expected to work.

Microsoft's LINQ to Objects

8 failures, all expected:

- Contains delegates to the ICollection<T>.Contains implementation if it exists, rather than using the default comparer for the type. This is a design and documentation issue which I've discussed in more detail in the Contains part of this series.
- Optimization: ElementAt and ElementAtOrDefault don't validate the specified index eagerly when the input sequence implements ICollection<T> but not IList<T>.
- Optimization: OfType always uses an intermediate iterator even when the input sequence already implements IEnumerable<T> and T is a non-nullable value type.
- Optimization: SequenceEqual doesn't compare the counts of the sequences eagerly even when both sequences implement ICollection<T>
- Correctness: OrderByDescending doesn't work if you use a key comparer which returns int.MinValue
- Consistency: Single and SingleOrDefault (with a predicate) don't throw InvalidOperationException as soon as they encounter a second element matching the predicate; the predicate-less overloads *do* throw as soon as they see a second element.

All of these have been discussed already, so I won't go into them now.

LinqBridge

LinqBridge had a total of 33 failures. I haven't looked into them in detail, but just going from the test output I've broken them down into the following broad categories:

- Optimization:
  - Cast never returns the original source, presumably always introducing an intermediate iterator.
  - All three of Microsoft's "missed opportunities" listed above are also missed in LinqBridge
- Use of input sequences:
  - Except and Intersect appear to read the first sequence first (possibly completely?) and then the second sequence. Edulinq and LINQ to Objects read the second sequence completely and then stream the first

sequence. This behaviour is undocumented.
    - Join, GroupBy and GroupJoin appear not to be deferred at all. If I'm right, this is a definite bug.
    - Aggregation accuracy: both Average and Sum over an IEnumerable<float> appear to use a float accumulator instead of a double. This is probably worth fixing for the sake of both range and accuracy, but isn't specified in the documentation.
    - OrderBy (etc) appears to apply the key selector multiple times while sorting. The behaviour here isn't documented, but as I mentioned before, it could produce performance issues unnecessarily.
- Exceptions:
    - ToDictionary should throw an exception if you give it duplicate keys; it appears not to - at least when a custom comparer is used. (It's possible it's just not passing the comparer along.)
    - The generic Max and Min methods don't return the null value for the element type when that type is nullable. Instead, they throw an exception - which is the normal behaviour if the element type is non-nullable. This behaviour isn't well documented, but is consistent with the behaviour of the non-generic overloads. See the Min/Max post for more details.
- General bugs:
    - The generic form of Min/Max appears not to ignore null values when the element type is nullable.
    - OrderByDescending appears to be broken in the same way as Microsoft's implementation
    - Range appears to be broken around its boundary testing.
    - Join, GroupJoin, GroupBy and ToLookup break when presented with null keys

Mono 4.0 (and 3.5, effectively)

Mono failed 18 of the tests. There are fewer definite bugs than in LinqBridge, but it's definitely not perfect. Here's the breakdown:

- Optimization:
    - Mono misses the same three opportunities that LinqBridge and Microsoft miss.
    - Contains(item) delegates to ICollection<T> when it's implemented, just like in the Microsoft implementation. (I assume the authors would call this an "optimization", hence its location in this section.) I believe that LinqBridge has the same behaviour, but that test didn't run in the LinqBridge configuration as it uses HashSet.
- Average/Sum accumulator types:
    - Mono appears to use float when working with float values, leading to

more accumulator error than is necessary.
- Average overflow for integer types
  - Mono appears to use checked arithmetic when *summing* a sequence, but not when taking the *average* of a sequence. So the average of { long.MaxValue, long.MaxValue, 2 } is 0. (This originally confused me into thinking it was using floating point types during the summation, but I now believe it's just a checked/unchecked issue.)
- Bugs:
  - Count doesn't overflow either with or without a predicate
  - The Max handling of double.NaN isn't in line with .NET. I haven't investigated the reason for this yet.
  - OrderByDescending is broken in the same way as for LinqBridge and the Microsoft implementation.
  - Range is broken for both Range(int.MinValue, 0) and Range(int.MaxValue, 1). Test those boundary cases, folks :)
  - When reversing a list, Mono *doesn't* buffer the current contents. In other words, changes made while iterating over the reversed list are visible in the returned sequence. The documentation isn't very clear about the desired behaviour here, admittedly.
  - GroupJoin and Join match null keys, unlike Microsoft's implementation.

How does Edulinq fare against other unit tests?

It didn't seem fair to *only* test other implementations against the Edulinq tests. After all, it's only natural that my tests should work against my own code. What happens if we run the Mono and LinqBridge tests against my code?

The LinqBridge tests didn't find anything surprising. There were two failures:

- I don't have the "delegate Contains to ICollection<T>.Contains" behaviour, which the tests check for.
- I don't optimize First in the case of the collection implementing IList<T>. I view this as a pretty dubious optimization to be honest - I doubt that creating an iterator to get to the first item is going to be much slower than checking for IList<T>, fetching the count, and then fetching the first item via the indexer... and it means that all *non-list* implementations also have to check whether the sequence implements IList<T>. I don't intend to change Edulinq for this.

The Mono tests picked up the same two failures as above, and two *genuine* bugs:

- By implementing Take via TakeWhile, I was iterating too far: in order for the condition to become false, we had to iterate to the first item we *wouldn't* return.
- ToLookup didn't accept null keys - a fault which propagated to GroupJoin, Join

and GroupBy too. (EDIT: It turns out that it's more subtle than that. Nothing should break, but the MS implementation *ignores* null keys for Join and GroupJoin. Edulinq now does the same, but I've raised a Connect issue to suggest this should *at least* be documented.)

I've fixed these in source control, and will add an addendum to each of the relevant posts (Take, ToLookup) when I have a moment spare.

There's one additional failure, trying to find the average of a sequence of two Int64.MaxValue values. That overflows on both Edulinq and LINQ to Objects - that's the downside of using an Int64 to sum the values. As mentioned, Mono suffers a degree of inaccuracy instead; it's all a matter of trade-offs. (A *really* smart implementation might use Int64 while possible, and then go up to using Double where necessary, I suppose.)

Unfortunately I don't have the tests for the Microsoft implementation, of course... I'd love to know whether there's anything I've failed with there.

Conclusion

This was very interesting - there's a mixture of failure conditions around, and plenty of "non-failures" where each implementation's tests are enforcing their own behaviour.

I do find it amusing that all three of the "mainstream" implementations have the same OrderByDescending bug though. Other than that, the clear bugs between Mono and LinqBridge don't intersect, which is slightly surprising.

It's nice to see that despite not setting out to create a "production-quality" implementation of LINQ to Objects, that's *mostly* what I've ended up with. Who knows - maybe some aspects of my implementation or tests will end up in Mono in the future :)

Given the various different optimizations mentioned in this post, I think it's only fitting that next time I'll discuss where we *can* optimize, where it's *worth* optimizing, and some more tricks we could still pull out of the bag...

Back to the table of contents.

Part 40 - Optimization

I'm not an expert in optimization, and most importantly I don't have any real-world benchmarks to support this post, so please take it with a pinch of salt. That said, let's dive into what optimizations are available in LINQ to Objects.

What do we mean by optimization?

Just as we think of refactoring as changing the internal structure of code without changing its externally visible behaviour, optimization is the art/craft/science/voodoo of changing the *performance* of code without changing its externally visible behaviour. Sort of.

This requires two definitions: "performance" and "externally visible behaviour". Neither are as simple as they sound. In almost all cases, performance is a trade-off, whether in speed vs memory, throughput vs latency, big-O complexity vs the factors *within* that complexity bound, and best case vs worst case.

In LINQ to Objects the "best case vs worst case" is the balance which we need to consider most often: in the cases where we can make a saving, how significant is that saving? How much does it cost in *every* case to make a saving *some* of the time? How often do we actually take the fast path?

Externally visible behaviour is even harder to pin down, because we definitely don't mean *all* externally visible behaviour. Almost all the optimizations in Edulinq are visible if you work hard enough - indeed, that's how I have unit tests for them. I can test that Count() uses the Count property of an ICollection<T> instead of iterating over it by creating an ICollection<T> implementation which works with Count but throws an exception if you try to iterate over it. I don't think we care about that sort of change to externally visible behaviour.

What we really mean is, "If we use the code in a *sensible* way, will we get the same results with the optimized code as we would without the optimization?" Much better. Nothing woolly about the term "sensible" at all, is there? More realistically, we could talk about a system where every type adheres to the contracts of every interface it implements - that would at least get rid of the examples used for unit testing. Still, even the performance of a system is externally visible... it's easy to tell the difference between an implementation of Count() which is optimized and one which isn't, if you've got a list of 10 million items.

How can we optimize in LINQ to Objects?

Effectively we have one technique for significant optimization in LINQ to Objects: finding out that a sequence implements a more capable interface than IEnumerable<T>, and then using that interface. (Or in the case of my optimization for HashSet<T> and Contains, using a concrete type in the same way.) Count() is the most obvious example of this: if the sequence implements ICollection or ICollection<T>, then we can use the Count property on that interface and we're done. No need to iterate at all.

These are generally good optimizations because they allow us to transform an O(n) computation into an O(1) computation. That's a pretty big win when it's applicable, and the cost of checking is *reasonably* small. So long as we hit the right type once every so often - and particularly if in those cases the sequences are long - then it's a net gain. The performance characteristics are likely to be reasonably fixed here for any one program - it's not like we'll *sometimes* win and *sometimes* lose for a specific query... it's that for some queries we win and some we won't. If we were micro-optimizing, we might want a way of calling a "non-optimized" version which didn't even bother trying the optimization, because we know it will always fail. I would regard such an approach as a colossal waste of effort in the majority of cases.

Some optimizations are slightly less obvious, particularly because they don't offer a change in the big-O complexity, but can still make a significant difference. Take ToArray, for example. If we know the sequence is an IList<T> we can construct an array of exactly the right size and ask the list to copy the elements into it. Chances are that copy can be very efficient indeed, basically copying a whole block of bits from one place to another - and we know we won't need any resizing. Compare that with building up a buffer, resizing periodically including copying all the elements we've already discovered. Every part of that process is going to be slower, but they're both O(n) operations really. This is a good example of where big-O notation doesn't tell the whole story. Again, the optimization is almost certainly a good one to make.

Then there are distinctly dodgy optimizations which *can* make a difference, but are unlikely to apply. My optimization for ElementAt and ElementAtOrDefault comes into play here. It's fine to check whether an object implements IList<T>, and use the indexer if so. That's an obvious win. But I have an extra optimization to exit quickly if we can find out that the given index is out of the bounds of the sequence. Unfortunately that optimization is only useful when:

- The sequence implements ICollection<T> or ICollection (but remember it has to implement IEnumerable<T> - there aren't many collections implementing *only* the non-generic ICollection, but the generic IEnumerable<T>)
- The sequence *doesn't* implement IList<T> (which gets rid of almost all

implementations of ICollection<T>)
- The given index is *actually* greater than or equal to the size of the collection

All that comes at the cost of a couple of type checks... not a great cost, and we *do* potentially save an O(n) check for being given an index out of the bounds of the collection... but how often are we really going to make that win? This is where I'd love to have something like Dapper, but applied to LINQ to Objects and running in a significant number of real-world projects, just logging in as light a way as possible how often we win, how often we lose, and how big the benefit is.

Finally, we come to the optimizations which don't make sense to me... such as the optimization for First in both Mono and LinqBridge. Both of these projects check whether the sequence is a list, so that they check the count and then use the indexer to fetch item 0 instead of calling GetEnumerator()/MoveNext()/Current. Now yes, there's a chance this avoids creating an extra object (although not always, as we've seen before) - but they're both O(1) operations which are likely to be darned fast. At this point not only is the payback very small (if it even exists) but the whole operation is likely to be so fast that the tiny check for whether the object implements IList<T> is likely to become more significant. Oh, and then there's the extra code complexity - yes, that's only relevant to the implementers, but I'd personally rather they spent their time on other things (like getting OrderByDescending to work properly... smirk). In other words, I think this is a *bad* target for optimization. At some point I'll try to do a quick analysis of just how often the collection has to implement IList<T> in order for it to be worth doing this - and whether the improvement is even measurable.

Of course there are other micro-optimizations available. When we don't need to fetch the current item (e.g. when skipping over items) let's just call MoveNext() instead of also assigning the return value of a property to a variable. I've done that in various places in Edulinq, but *not* as an optimization strategy, which I suspect won't make a significant difference, but for readability - to make it clearer to the reader that we're just moving along the iterator, not examining the contents as we go.

The only other piece of optimization I think I've performed in Edulinq is the "yield the first results before sorting the rest" part of my quicksort implementation. I'm reasonably proud of that, at least conceptually. I don't think it really fits into any other bucket - it's just a matter of thinking about what we really need and when, deferring work just in case we never need to do it.

What can we *not* optimize in LINQ to Objects?

I've found a few optimizations in both Edulinq and other implementations which I believe to be invalid.

Here's an example I happened to look at just this morning, when reviewing the code for Skip:

```
var list = source as IList<TSource>;
if (list != null)
{
    count = Math.Max(count, 0);
    // Note that "count" is the count of items to skip
    for (int index = count; index < list.Count; index++)
    {
        yield return list[index];
    }
    yield break;
}
```

If our sequence is a list, we can just skip straight to the right part of it and yield the items one at a time. That sounds great, but what if the list changes (or is even truncated!) while we're iterating over it? An implementation working with the simple iterator would usually throw an exception, as the change would invalidate the iterator. This is definitely a behavioural change. When I first wrote about Skip, I included this as a "possible" optimization - and actually turned it on in the Edulinq source code. I now believe it to be a mistake, and have removed it completely.

Another example is Reverse, and how it should behave. The documentation is fairly unclear, but when I ran the tests, the Mono implementation used an optimization whereby if the sequence is a list, it will just return items from the tail end using the indexer. (This has now been fixed - the Mono team is quick like that!) Again, that means that changes made to the list while iterating will be reflected in the reversed sequence. I believe the documentation for Reverse *ought* to be clear that:

- Execution is deferred: the input sequence isn't read when the method is called.
- When the result sequence is first read by the caller, a snapshot is taken, and *that's* what's used to return the data.
- If the result sequence is read more than once (i.e. GetEnumerator is called more than once) then a new snapshot is created each time - so changes to the input sequence between calls to GetEnumerator on the result sequence *will* be observed.

Now this is still not as precise as it might be in terms of what "reading" a sequence entails - in particular, a simple implementation of Reverse (as per Edulinq) will actually take the snapshot on the first call to MoveNext() on the iterator returned by GetEnumerator() - but that's probably not too bad. The snapshotting behaviour itself is important though, and should be made explicit in my opinion.

The problem with both of these "optimizations" is arguably that they're applying list-

based optimizations *within an iterator block used for deferred execution*. Optimizing for lists either upfront at the point of the initial method call or within an immediate execution operator (Count, ToList etc) is fine, because we assume the sequence won't change during the course of the method's execution. We can't make that assumption with an iterator block, because the flow of the code is very different: our code is visited repeatedly based on the caller's use of MoveNext().

Sequence identity

Another aspect of behaviour which isn't well-specified is that of identity. When is it valid for an operator to return the input sequence itself as the result sequence?

In the Microsoft implementation, this can occur in two operators: AsEnumerable (which *always* returns the input sequence reference, even if it's null) and Cast (which returns the original reference only if it actually implements IEnumerable<TResult>).

In Edulinq, I have two other operators which can return the input sequence: OfType (only if the original reference implements IEnumerable<TResult> and TResult is a non-nullable value type) and Skip (if you provide a count which is zero or negative). Are these valid optimizations? Let's think about why we might *not* want them to be...

If you're returning a sequence from one layer of your code to another, you usually want that sequence to be viewed *only* as a sequence. In particular, if it's backed by a List<T>, you don't want callers casting to List<T> and modifying the list. With any operator implemented by an iterator block, that's fine - the object returned from the operator has no accessible reference to its input, and the type itself only implements IEnumerable<T> (and IEnumerator<T>, and IDisposable, etc - but not IList<T>). It's not so good if the operator decides it's okay to return the original reference.

The C# language specification refers to this in the section about query expression translation: a no-op projection at the end of a query can be omitted *if and only if there are other operators in the query*. So a query expression of "from foo in bar select foo" will translate to "bar.Select(foo => foo)" but if we had a "where" clause in the query, the Select call would be removed. It's worth noting that the call to "Cast" generated when you explicitly specify the type of a range variable is *not* enough to prevent the "no-op" projection from being generated... it's almost as if the C# team "knows" that Cast can leak sequence identity whereas Where can't.

Personally I think that the "hiding" of the input sequence should be guaranteed where it makes sense to do so, and explicitly *not* guaranteed otherwise. We could also add an operator of something like "HideIdentity" which would simply (and unconditionally) add an extra iterator block into the pipeline. That way library authors wouldn't have to guess, and would have a clear way of expressing their intention. Using Select(x => x)

or Skip(0) is *not* clear, and in the case of Skip it would even be pointless when using Edulinq.

As for whether my optimizations are valid - that's up for debate, really. It seems hard to justify why leaking sequence identity would be okay for Cast but *not* okay for OfType, whereas I think there's a better case for claiming that Skip should always hide sequence identity.

The Contains issue...

If you remember, I have a disagreement around what Contains should do when you don't provide an equality comparer, and when the sequence implements ICollection<T>. I believe it should be consistent with the rest of LINQ to Objects, which *always* uses the default equality comparer for the element type when it needs one but the user hasn't specified one. Everyone else (Microsoft, Mono, LinqBridge) has gone with delegating to the collection's implementation of ICollection<T>.Contains. That plays well in terms of consistency of what happens if you call Contains on that object, so that it doesn't matter what the compile-time type is. That's a debate to go into in another post, but I just want to point out that this is *not* an example of optimization. In some cases it may be faster (notably for HashSet<T>) but it stands a *very* good chance of changing the behaviour. There is absolutely nothing to suggest that the equality comparer used by ICollection<T> should be the default one for the type - and in some cases it definitely isn't.

It's therefore a matter of what *result* we want to get, not how to get that result faster. It's correctness, not optimization - but both the LinqBridge and Mono tests which fail for Edulinq are called "Contains_CollectionOptimization_ReturnsTrueWithoutEnumerating" - and I think that shows a mistaken way of thinking about this.

Can we go further?

I've been considering a couple of optimizations which I believe to be perfectly legitimate, but which none of the implementations I've seen have used. One reason I haven't implemented them myself yet is that they will reduce the effectiveness of all my unit tests. You see, I've generally used Enumerable.Range as a good way of testing a non-list-based sequence... but what's to stop Range and Repeat being implemented *as* IList<T> implementations?

All the non-mutating members are easy to implement, and we can just throw exceptions from the mutating members (as other read-only collections do).

Would this be more efficient? Well yes, if you ever performed a Count(), ElementAt(),

ToArray(), ToList() etc operation on a range or a repeated element... but how often is *that* going to happen? I suspect it's pretty rare - probably rare enough not to make it worth my time, particularly when you then consider all the tests that would have to be rewritten to use something other than Range when I wanted a non-list sequence...

Conclusion

Surprise, surprise - doing optimization well is difficult. When it's obvious what *can* be done, it's not obvious what *should* be done... and sometimes it's not even what is valid in the first place.

Note that none of this has really talked about data structures and algorithms. I looked at some options when implementing ordering, and I'm *still* thinking about the best approach for implementing TopBy (probably either a heap or a self-balancing tree - something which could take advantage of the size being constant would be nice) - but *in general* the optimizations here haven't required any cunning knowledge of computer science. That's quite a good thing, because it's many years since I've studied CS seriously...

I suspect that with this post more than almost any other, I'm likely to want to add extra items in the future (or amend mistakes which reveal my incompetence). Watch this space.

Next up, I think it would be worth revisiting query expressions from scratch. Anyone who's read C# in Depth or has followed this blog for long enough is likely to be able to skip it, but I think the series would be incomplete without a quick dive into the compiler translations involved.

Back to the table of contents.

Part 41 - How query expressions work

Okay, first a quick plug. This *won't* be in as much detail as chapter 11 of C# in Depth. If you want more, buy a copy. (Until Feb 1st, there's 43% off it if you buy it from Manning with coupon code j2543.) Admittedly that chapter has to *also* explain all the basic operators rather than just query expression translations, but that's a fair chunk of it.

If you're already familiar with query expressions, don't expect to discover anything particularly insightful here. However, you might be interested in the cheat sheet at the end, in case you forget some of the syntax occasionally. (I know I do.)

What is this "query expression" of which you speak?

Query expressions are a little nugget of goodness hidden away in section 7.16 of the C# specification. Unlike some language features like generics and dynamic typing, query expressions keep themselves to themselves, and don't impinge on the rest of the spec. A query expression is a bit of C# which looks a bit like a mangled version of SQL. For example:

```
from person in people
where person.FirstName.StartsWith("J")
orderby person.Age
select person.LastName
```

It looks somewhat unlike the rest of C#, which is both a blessing and a curse. On the one hand, queries stand out so it's easy to see they're queries. On the other hand... they stand out rather than fitting in with the rest of your code. To be honest I haven't found this to be an issue, but it can take a little getting used to.

Every query expression can be represented in C# code, but the reverse isn't true. Query expressions only take in a subset of the standard query operators - and only a limited set of the overloads, at that. It's not unusual to see a query expression followed by a "normal" query operator call, for example:

```
var list = (from person in people
            where person.FirstName.StartsWith("J")
            orderby person.Age
            select person.LastName)
           .ToList();
```

So, that's very roughly what they look like. That's the sort of thing I'm dealing with in this post. Let's start dissecting them.

Compiler translations

First it's worth introducing the general principle of query expressions: they effectively get translated step by step into C# which eventually *doesn't* contain any query expressions. To stick with our first example, that ends up be translated into this code:

```
people.Where(person => person.FirstName.StartsWith("J"))
      .OrderBy(person => person.Age)
      .Select(person => person.LastName)
```

It's important to understand that the compiler hasn't done anything apart from systematic translation to get to this point. In particular, so far we haven't depended on what "people" is, nor "Where", "OrderBy" or "Select".

Can you tell what this code does yet? You can probably hazard a pretty good guess,

but you can't tell. Is it going to call Edulinq.Enumerable.Select, or System.Linq.Enumerable.Select, or something entirely different? It depends on the context. Heck, "people" could be the name of a type which has a static Where method. Or maybe it could be a reference to a class which has an *instance* method called Where... the options are open.

Of course, they don't stay open for long: the compiler takes that expression and compiles it applying all the normal rules. It converts the lambda expression into either a delegate or an expression tree, tries to resolve Where, OrderBy and Select as normal, and life continues. (Don't worry if you're not sure about expression trees yet - I'll come to them in another post.)

The important point is that the query expression translations don't know about System.Linq. The spec barely mentioned IEnumerable<T>, and certainly doesn't rely on it. The whole thing is *pattern based*. If you have an API which provides some or all of the operators used by the pattern, in an appropriate way, you can use query expressions with it. That's the secret sauce that allows you to use the same syntax for LINQ to Objects, LINQ to SQL, LINQ to Entities, Reactive Extensions, Parallel Extensions and more.

Range variables and the initial "from" clause

The first part of the query to look at is the first "from" clause, at the start of the query. It's worth mentioning upfront that this is handled somewhat differently to any *later* "from" clauses - I'll explain how they're translated later.

So we have an expression of the form:

```
from [type] identifier in expression
```

The "expression" part is just any expression. In most cases there *isn't* a type specified, in which case the translated version is simply the expression, but with the compiler remembering the identifier as a *range variable*. I'll do my best to explain what range variables are in a minute :)

If there *is* a type specified, that represents a call to Cast<*type*>(). So examples of the two translations so far are:

```
// Query (incomplete)
from x in people

// Translation (+ range variable of "x")
people


// Query (incomplete)
from Person x in people

// Translation (+ range variable of "x")
(people).Cast<Person>()
```

These aren't complete query expressions - queries have very precise rules about how they can start and end. They *always* start with a "from" clause like this, and always end either with a "group by" clause or a "select" clause.

So what's the point of the range variable? Well, that's what gets used as the name of the lambda expression parameter used in all the later clauses. Let's add a select clause to create a complete expression and demonstrate how the variable could be used.

A "select" clause

A select clause is usually translated into a call to Select, using the "body" of the

clause as the body of the lambda expression... and the range variable as the parameter. So to expand our previous query, we might have this translation:

```
// Query
from x in people
select x.Name

// Translation
people.Select(x => x.Name)
```

That's all that range variables are used for: to provide placeholders within lambda expressions, effectively. They're quite unlike normal variables in most senses. It only makes sense to talk about the "value" of a range variable within a particular clause at a particular point in time when the clause is executing, for one particular value. Their nearest conceptual neighbour is probably the iteration variable declared in a foreach statement, but even that's not really the same - particularly given the way iteration variables are captured, often to the surprise of developers.

The body part has to be a single *expression* - you can't use "statement lambdas" in query expressions. For example, there's no query expression which would translate to this:

```
// Can't express this in a query expression
people.Select(x => {
                    Console.WriteLine("Got " + x);
                    return x.Name;
                  })
```

That's a perfectly valid C# expression, it's just there's now way of expressing it directly as a query expression.

I mentioned that a select clause *usually* translates into a Select call. There are two cases where it doesn't:

- If it's the sole clause after a secondary "from" clause, or a "group by", "join" or "join ... into" clause, the body is used in the translation of that clause
- If it's an "identity" projection coming after another clause, it's removed entirely.

I'll deal with the first point when we reach the relevant clauses. The second point leads to these translations:

```
// Query
from x in people
where x.IsAdult
select x

// Translation: Select is removed
people.Where(x => x.IsAdult)


// Query
from x in people
select x

// Translation: Select is *not* removed
people.Select(x => x)
```

The point of including the "pointless" select in the second translation is to hide the original source sequence; it's assumed that there's no need to do this in the first translation as the "Where" call will already have protected the source sufficiently.

The "where" clause

This one's really simple - especially as we've already seen it! A where clause always just translates into a Where call. Sample translation, this time with no funny business

removing degenerate query expressions:

```
// Query
from x in people
where x.IsAdult
select x.Name

// Translation
people.Where(x => x.IsAdult)
      .Select(x => x.Name)
```

Note how the range variable is propagated through the query.

The "orderby" clause

Here's a secret: I can never remember offhand whether it's "orderby" or "order by" - it's confusing because it really is "group by", but "orderby" is actually just a single word. Of course, Visual Studio gives a pretty unsubtle hint in terms of colouring.

In the simplest form, an orderby clause might look like this:

```
// Query
from x in people
orderby x.Age
select x.Name

// Translation
people.OrderBy(x => x.Age)
      .Select(x => x.Name)
```

There are two things which can add complexity though:

- You can order by multiple expressions, separating them by commas
- Each expression can be ordered ascending implicitly, ascending *explicitly* or descending explicitly.

The first sort expression is always translated into OrderBy or OrderByDescending; subsequent ones always become ThenBy or ThenByDescending. It makes no difference whether you explicitly specify "ascending" or not - I've very rarely seen it in real queries. Here's an example putting it all together:

```
// Query
from x in people
orderby x.Age, x.FirstName descending, x.LastName ascending
select x.LastName

// Translation
people.OrderBy(x => x.Age)
      .ThenByDescending(x => x.FirstName)
      .ThenBy(x => x.LastName)
      .Select(x => x.LastName)
```

Top tip: don't use multiple "orderby" clauses consecutively. This query is almost certainly *not* what you want:

```
// Don't do this!
from x in people
orderby x.Age
orderby x.FirstName
select x.LastName
```

That will end up sorting by FirstName and *then* Age, and doing so rather slowly as it has to sort twice.

The "group by" clause

Grouping is another alternative to "select" as the final clause in a query. There are two expressions involved: the element selector (what you want to get in each group) and the key selector (how you want the groups to be organized). Unsurprisingly, this uses the GroupBy operator. So you might have a query to group people in families by their last name, with each group containing the first names of the family members:

```
// Query expression
from x in people
group x.FirstName by x.LastName

// Translation
people.GroupBy(x => x.LastName, x => x.FirstName)
```

If the element selector is trivial, it isn't specified as part of the translation:

```
// Query expression
from x in people
group x by x.LastName

// Translation
people.GroupBy(x => x.LastName)
```

Query continuations

Both "select" and "group by" can be followed by "into *identifier*". This is known as a *query continuation*, and it's really simple. Its translation in the specification isn't in terms of a method call, but instead it transforms one query expression into another, effectively nesting one query as the source of another. I find that translation tricky to think about, personally... I prefer to think of it as using a temporary variable, like this:

```
// Original query
var query = from x in people
            select x.Name into y
            orderby y.Length
            select y[0];

// Query continuation translation
var tmp = from x in people
          select x.Name;

var query = from y in tmp
            orderby y.Length
            select y[0];

// Final translation into methods
var query = people.Select(x => x.Name)
                  .OrderBy(y => y.Length)
                  .Select(y => y[0]);
```

Obviously that final translation *could* have been expressed in terms of two statements as well... they'd be equivalent. This is why it's important that LINQ uses deferred execution - you can split up a query as much as you like, and it won't alter the execution flow. The query wouldn't actually *execute* when the value is assigned to "tmp" - it's just *preparing* the query for execution.

Transparent identifiers and the "let" clause

The rest of the query expression clauses all introduce an extra range variable in some form or other. This is the part of query expression translation which is hardest to understand, because it affects how any usage of the range variable in the query expression is translated.

We'll start with probably the simplest of the remaining clauses: the "let" clause. This

simply introduces a new range variable based upon a projection. It's a bit like a "select", but after a "let" clause both the original range variable *and* the new one are in scope for the rest of the query. They're typically used to avoid redundant computations, or simply to make the code simpler to read. For example, suppose computing an employee's tax is a complicated operation, and we want to display a list of employees and the tax they pay, with the higher tax-payer first:

```
from x in employees
let tax = x.ComputeTax()
orderby tax descending
select x.LastName + ": " + tax
```

That's pretty readable, and we've managed to avoid computing the tax twice (once for sorting and once for display).

The problem is, both "x" and "tax" are in scope at the same time... so what are we going to pass to the Select method at the end? We need one entity to pass through our query, which knows the value of both "x" and "tax" at any point (after the "let" clause, obviously). This is precisely the point of a transparent identifier. You can think of the above query as being translated into this:

```
// Translation from "let" clause to another query expression
from x in employees
select new { x, tax = x.ComputeTax() } into z
orderby z.tax descending
select z.x.LastName + ": " + z.tax

// Final translated query
employees.Select(x => new { x, tax = x.ComputeTax() })
         .OrderByDescending(z => z.tax)
         .Select(z => z.x.LastName + ": " + z.tax)
```

Here "z" is the transparent identifier - which I've made somewhat more opaque by giving it a name. In the specification, the query translations are performed in terms of "*" - which clearly isn't a valid identifier, but which stands in for the transparent one.

The good news about transparent identifiers is that most of the time you don't need to think of them at all. They simply let you have multiple range variables in scope at the same time. I find myself only bothering to think about them explicitly when I'm trying to work out the full translation of a query expression which uses them. It's worth knowing about them to avoid being stumped by the concept of (say) a select clause being able to use multiple range variables, but that's all.

Now that we've got the basic concept, we can move onto the final few clauses.

Secondary "from" clauses

We've seen that the introductory "from" clause isn't actually translated into a method call, but any subsequent ones are. The syntax is still the same, but the translation uses SelectMany. In many cases this is used just like a cross-join (Cartesian product) but it's more flexible than that, as the "inner" sequence introduced by the secondary "from" clause can depend on the current value from the "outer" sequence. Here's an example of that. with the call to SelectMany in the translation:

```
// Query expression
from parent in adults
from child in parent.Children
where child.Gender == Gender.Male
select child.Name + " is a son of " + parent.Name

// Translation (using z for the transparent identifier)
adults.SelectMany(parent => parent.Children,
                  (parent, child) => new { parent, child })
      .Where(z => z.child.Gender == Gender.Male)
      .Select(z => z.child.Name + " is a son of " + z.parent.Name;
```

Again we can see the effect of the transparent identifier - an anonymous type is introduced to propagate the { parent, child } tuple through the rest of the query.

There's a special case, however - if "the rest of the query" is *just* a "select" clause, we don't need the anonymous type. We can just apply the projection directly in the SelectMany call. Here's a similar example, but this time without the "where" clause:

```
// Query expression
from parent in adults
from child in parent.Children
select child.Name + " is a child of " + parent.Name

// Translation (using z for the transparent identifier)
adults.SelectMany(parent => parent.Children,
                  (parent, child) => child.Name + " is a child of " +
parent.Name)
```

This same trick is used in GroupJoin and Join, but I won't go into the details there. It's simpler to just provide examples which use the shortcut, instead of including unnecessary extra clauses just to force the transparent identifier to appear in the translation.

Note that just like the introductory "from" clause, you can specify a type for the range variable, which forces a call to "Cast<>".

Simple "join" clauses (no "into")

A "join" clause without an "into" part corresponds to a call to the Join method, which represents an inner equijoin. In some ways this is like an extra "from" clause with a "where" clause to provide the relevant filtering, but there's a significant difference: while the "from" clause (and SelectMany) allow you to project each element in the outer sequence to an inner sequence, in Join you merely provide the inner sequence directly, once. You also have to specify the two key selectors - one for the outer sequence, and one for the inner sequence. The general syntax is:

```
join identifier in inner-sequence on outer-key-selector equals inner-key-
selector
```

The identifier names the extra range variable introduced. Here's an example including the translation:

```
// Query expression
from customer in customers
join order in orders on customer.Id equals order.CustomerId
select customer.Name + ": " + order.Price

// Translation
customers.Join(orders,
               customer => customer.Id,
               order => order.CustomerId,
               (customer, order) => customer.Name + ": " + order.Price)
```

Note how if you put the key selectors the wrong way round, it's highly unlikely that the result will compile - the lambda expression for the outer sequence doesn't "know about" the inner sequence element, and vice versa. The C# compiler is even nice enough to guess the probable cause, and suggest the fix.

Group joins - "join ... into"

Group joins look exactly the same as inner joins, except they have an extra "into *identifier*" part at the end. Again, this introduces an extra range variable - but it's the identifier after the "into" which ends up in scope, not the one after "join"; that one is

only used in the key selector. This is easier to see when we look at a sample translation:

```
// Query expression
from customer in customers
join order in orders on customer.Id equals order.CustomerId into
customerOrders
select customer.Name + ": " + customerOrders.Count()

// Translation
customers.GroupJoin(orders,
                    customer => customer.Id,
                    order => order.CustomerId,
                    (customer, customerOrders) => customer.Name + ": " +
customerOrders.Count())
```

If we had tried to refer to "order" in the select clause, the result would have been an error: it's not in scope any more. Note that this is *not* a query continuation unlike "select ... into" and "group ... into". It introduces a new range variable, but all the previous range variables are still in scope.

That's it! That's all the translations that the C# compiler supports. VB's query expressions are rather richer - but I suspect that's at least *partly* because it's more painful to write the "dot notation" syntax in VB, as the lambda expression syntax isn't as nice as C#'s.

Translation cheat sheet

I thought it would be useful to produce a short table of the kinds of clauses supported in query expressions, with the translation used by the C# compiler. The translation is given assuming a single range variable named "x" is in scope. I haven't given the alternative options where transparent identifiers are introduced - this table isn't meant to be a replacement for all the information above! (Likewise this doesn't mention the optimizations for degenerate query expressions or "identity projection" groupings.)

Query expression clause	**Translation**
First "from *[type] x* in *sequence*"	Just "sequence" or "sequence.Cast<type>()", but with the introduction of a range variable
Subsequent "from" clauses: "from *[type] y* in *projection*"	SelectMany(x => projection, (x, y) => new { x, y }) or SelectMany(x => projection.Cast<type>(), (x, y) => new { x, y })
where *predicate*	Where(x => predicate)
select *projection*	Select(x => projection)
let y = projection	Select(x => new { x, y = projection })
orderby o1, o2 ascending, o3 descending (Each ordering may have descending or ascending specified explicitly; the default is ascending)	OrderBy(x => o1) .ThenBy(x => o2) .ThenByDescending(x => o3)
group *projection* by *key-selector*	GroupBy(x => key-selector, x => projection)
join *y* in *inner-sequece* on *outer-key-selector* equals *inner-key-selector*	Join(x => outer-key-selector, y => inner-key-selector, (x, y) => new { x, y })
join *y* in *inner-sequece* on *outer-key-selector* equals *inner-key-selector* into *z*	GroupJoin(x => outer-key-selector, y => inner-key-selector, (x, z) => new { x, z })
query1 into *y* *query2*	(Translation in terms of a new query expression) from y in (query1) query2

Conclusion

Hopefully that's made a certain amount of sense out of a fairly complicated topic. I find it's one of those "aha!" things - at some point it clicks, and then seems reasonably simple (aside from transparent identifiers, perhaps). Until that time, query expressions can be a bit magical.

As an aside, I have a sneaking suspicion that one of my first blog posts consisted of my initial impressions of LINQ, written in a plane on the way to the MVP conference in Seattle in September 2005. I would check, but I'm finishing this post in another plane, this time on the way to San Francisco. I think I'd have been somewhat surprised to be told in 2005 that I'd still be writing blog posts about LINQ over five years later. Mind you, I can think of any number of things which have happened in the intervening years which would have astonished me to about the same degree.

Next time: some more thoughts on optimization. Oh, and I'm likely to update my wishlist of extra operators as well, but within the existing post.

Back to the table of contents.

Part 42 - More optimization

A few parts ago, I jotted down a few thoughts on optimization. Three more topics on that general theme have occurred to me, one of them prompted by the comments.

User-directed optimizations

I mentioned last time that for micro-optimization purposes, we could derive a tiny benefit if there were operators which allowed us to turn off potential optimizations - effectively declare in the LINQ query that we believed the input sequence would *never* be an IList<T> or an ICollection<T>, so it wasn't worth checking it. I still believe that level of optimization would be futile.

However, going the other way is entirely possible. Imagine if we could say, "There are probably a lot of items in this collection, and the operations I want to perform on them are independent and thread-safe. Feel free to parallelize them."

That's exactly what Parallel LINQ gives you, of course. A simple call to AsParallel() somewhere in the query - often at the start, but it doesn't have to be - enables parallelism. You need to be careful how you use this, of course, which is why it's opt-in... and it gives you a fair amount of control in terms of degrees of potential parallelism, whether the results are required in the original order and so on.

In some ways my "TopBy" proposal is similar in a very small way, in that it gives information relatively early in the query, allowing the subsequent parts (ThenBy clauses) to take account of the extra information provided by the user. On the other hand, the effect is extremely localized - basically just for the sequence of clauses to do with ordering.

Related to the idea of parallelism is the idea of side-effects, and how they affect LINQ to Objects itself.

Side-effects and optimization

The optimizations in LINQ to Objects appear to make some assumptions about side-effects:

- Iterating over a collection won't cause any side-effects
- Predicates *may* cause side-effects

Without the first point, all kinds of optimizations would effectively be inappropriate. As

the simplest example, Count() won't use an iterator - it will just take the count of the collection. What if this was an odd collection which mutated something during iteration, though? Or what if accessing the Count property itself had side-effects? At that point we'd be violating our principle of not changing observable behaviour by optimizing. Again, the optimizations are basically assuming "sensible" behaviour from collections.

There's a rather more subtle possible cause of side-effects which I've never seen discussed. In some situations - most obviously Skip - an operator can be implemented to move over an iterator for a time without taking each "current" value. This is due to the separation of MoveNext() from Current. What if we were dealing with an iterator which had side-effects *only when Current was fetched*? It would be easy to write such a sequence - but again, I suspect there's an implicit assumption that such sequences simply don't exist, or that it's reasonable for the behaviour of LINQ operators with respect to them to be left unspecified.

Predicates, on the other hand, might not be so sensible. Suppose we were computing "sequence.Last(x => 10 / x > 1)" on the sequence { 5, 0, 2 }. Iterating over the sequence forwards, we end up with a DivideByZeroException - whereas if we detected that the sequence was a list, and worked our way backwards from the end, we'd see that 10 / 2 > 1, and return that last element (2) immediately. Of course, exceptions aren't the only kind of side-effect that a predicate can have: it *could* mutate other state. However, it's generally easier to spot that and cry foul of it being a proper functional predicate than notice the possibility of an exception.

I believe this is the reason the predicated Last overload *isn't* optimized. It would be nice if these assumptions were documented, however.

Assumptions about performance

There's a final set of assumptions which the common ICollection<T>/IList<T> optimizations have all been making: that using the more "direct" members of the interfaces (specifically Count and the indexer) are more efficient than simply iterating. The interfaces make no such declarations: there's no requirement that Count *has* to be O(1), for example. Indeed, it's not even the case in the BCL. The first time you ask a "view between" on a sorted set for its count after the underlying set has changed, it has to count the elements again.

I've had this problem before, removing items from a HashSet in Java. The problem is that there's no way of communicating this information in a standardized way. We *could* use attributes for everything, but it gets very complicated, and I strongly suspect it would be a complete pain to use. Basically, performance is one area where abstractions just don't hold up - or rather, the abstractions aren't *designed* to

include performance characteristics.

Even if we knew the complexity of (say) Count that still wouldn't help us necessarily. Suppose it's an O(n) operation - that sounds bad, until you discover that for this particular horrible collection, *each iteration step* is also O(n) for some reason. Or maybe there's a collection with an O(1) count but a *horrible* constant value, whereas iterating is really quick per item... so for small values of O(n), iteration would be faster. Then you've got to bear in mind how much processor time is needed trying to work out the fastest approach... it's all bonkers.

So instead we make these assumptions, and for the most part they're correct. Just be aware of their presence.

Conclusion

I have reached the conclusion that I'm tired, and need sleep. I might write about Queryable, IQueryable and query expressions next time.

Back to the table of contents.

Part 43 - Out-of-process queries with IQueryable

I've been putting off writing about this for a while now, mostly because it's such a huge topic. I'm not going to try to give more than a brief introduction to it here - don't expect to be able to whip up your own LINQ to SQL implementation afterwards - but it's worth at least having an idea of what happens when you use something like LINQ to SQL, NHibernate or the Entity Framework.

Just as LINQ to Objects is primarily interested in IEnumerable<T> and the static Enumerable class, so out-of-process LINQ is primarily interested in IQueryable<T> and the static Queryable class... but before we get to them, we need to talk about expression trees.

Expression Trees

To put it in a nutshell, expression trees encapsulate logic in *data* instead of *code*. While you *can* introspect .NET code via MethodBase.GetMethodBody and then MethodBody.GetILAsByteArray, that's not really a practical approach. The types in the System.Linq.Expressions define expressions in an easier-to-process manner. When expression trees were introduced in .NET 3.5, they were strictly for *expressions*, but the Dynamic Language Runtime uses expression trees to represent operations, and the range of logic represented had to expand accordingly, to include things like blocks.

While you certainly *can* build expression trees yourself (usually via the factory methods on the nongeneric Expression class), and it's fun to do so at times, the most common way of creating them is to use the C# compiler's support for them via lambda expressions. So far we've always seen a lambda expression being converted to a delegate, but it can also convert lambdas to instances of Expression<TDelegate>, where TDelegate is a delegate type which is compatible with the lambda expression. A concrete example will help here. The statement:

```
Expression<Func<int, int>> addOne = x => x + 1;
```

will be compiled into code which is *effectively* something like this:

```
var parameter = Expression.Parameter(typeof(int), "x");
var one = Expression.Constant(1, typeof(int));
var addition = Expression.Add(parameter, one);
var addOne = Expression.Lambda<Func<int, int>>(addition,  new
ParameterExpression[] { parameter });
```

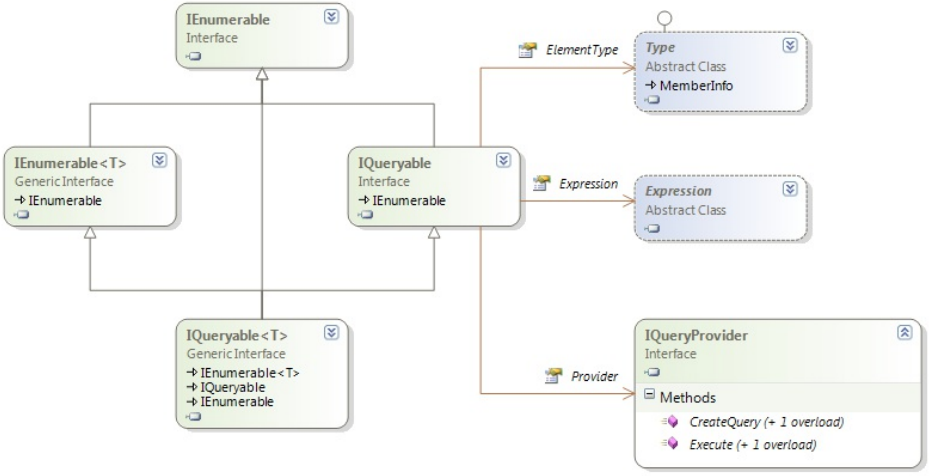
The compiler has some tricks up its sleeves which allow it to refer to methods, events and the like in a simpler way than we can from code, but largely you can regard the transformation as just a way of making life a *lot* simpler than if you had to build the expression trees yourself every time.

IQueryable, IQueryable<T> and IQueryProvider

Now that we've got the idea of being able to inspect logic relatively easily at execution time, let's see how it applies to LINQ.

There are three interfaces to introduce, and it's probably easiest to start with how

they appear in a class diagram:



Most of the time, queries are represented using the generic IQueryable<T> interface, but this doesn't actually add much over the nongeneric IQueryable interface it extends, other than *also* extending IEnumerable<T> - so you can iterate over the contents of an IQueryable<T> just as with any other sequence.

IQueryable contains the interesting bits, in the form of three properties: ElementType which indicates the type of the elements within the query (in other words, a dynamic form of the T from IQueryable<T>), Expression returns the expression tree for the query so far, and Provider returns the query provider which is responsible for creating *new* queries and executing the existing one. We won't need to use the ElementType property ourselves, but we'll need both the Provider and Expression properties.

The static Queryable class

We're not going to implement any of the interfaces ourselves, but I've got a small sample program to demonstrate how they all work, imagining we were implementing most of Queryable ourselves. This static class contains extension methods for IQueryable<T> just as Enumerable does for IEnumerable<T>. *Most* of the query operators from LINQ to Objects appear in Queryable as well, but there are a few notable omissions, such as the To{Lookup, Array, List, Dictionary} methods. If you call one of those on an IQueryable<T>, the Enumerable implementations will be used instead. (IQueryable<T> extends IEnumerable<T>, so the extension methods in Enumerable are applicable to IQueryable<T> sequences as well.)

The big difference between the Queryable and Enumerable methods in terms of their *declarations* is in the parameters:

- The "source" parameter in Queryable is always of type IQueryable<TSource> instead of IEnumerable<TSource>. (Other sequence parameters such as the sequence to concatenate for Queryable.Concat are expressed as IEnumerable<T>, interestingly enough. This allows you to express a SQL query using "local" data as well; the query methods work out whether the sequence is actually an IQueryable<T> and act accordingly.)

- Any parameters which were delegates in Enumerable are expression trees in Queryable; so while the selector parameter in Enumerable.Select is of type Func<TSource, TResult>, the equivalent in Queryable.Select is of type Expression<Func<TSource, TResult>>

The big difference between the methods in terms of what they *do* is that whereas the Enumerable methods actually do the work (eventually - possibly after deferred execution of course), the Queryable methods themselves really *don't* do any work: they just ask the query provider to build up a query indicating that they've been called.

Let's have a look at Where for example. If we wanted to implement Queryable.Where, we would have to:

- Perform argument checking
- Get the "current" query's Expression
- Build a new expression representing a call to Queryable.Where using the current expression as the source, and the predicate expression as the predicate
- Ask the current query's provider to build a new IQueryable<T> based on that call expression, and return it.

It all sounds a bit recursive, I realize - the Where call needs to record that a Where call has happened... but that's all. You may very well wonder where all the work is happening. We'll come to that.

Now building a call expression is slightly tedious because you need to have the right MethodInfo - and as Where is overloaded, that means distinguishing between the two Where methods, which is easier said than done. I've actually used a LINQ query to find the right overload - the one where the predicate parameter Expression<Func<T, bool>> rather than Expression<Func<T, int, bool>>. In the .NET implementation, methods can use MethodBase.GetCurrentMethod() instead... although equally they could have created a bunch of static variables computed at class initialization time. We can't use GetCurrentMethod() for experimentation purposes, because the query provider is likely to expect the exact correct method from System.Linq.Queryable in the System.Core assembly.

Here's our sample implementation, broken up quite a lot to make it easier to understand:

```
public static IQueryable<TSource> Where<TSource>(
    this IQueryable<TSource> source,
    Expression<Func<TSource, bool>> predicate)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
    if (predicate == null)
    {
        throw new ArgumentNullException("predicate");
    }

    Expression sourceExpression = source.Expression;
    Expression quotedPredicate = Expression.Quote(predicate);

    // This gets the "open" method, without specific type arguments. The
second parameter
    // of the method we want is of type Expression<Func<TSource, bool>>, so
the sole generic
    // type argument to Expression<T> itself has two generic type arguments.
    // Let's face it, reflection on generic methods is a mess.
    MethodInfo method = typeof(Queryable).GetMethods()
                                         .Where(m => m.Name == "Where")
```

```
                                        .Where(m => m.GetParameters()[1]
                                                     .ParameterType

.GetGenericArguments()[0]

.GetGenericArguments().Length == 2)
                                        .First();

    // This gets the method with the same type arguments as ours
    MethodInfo closedMethod = method.MakeGenericMethod(new Type[] {
typeof(TSource) });

    // Now we can create a *representation* of this exact method call
    Expression methodCall = Expression.Call(closedMethod, sourceExpression,
quotedPredicate);

    // ... and ask our query provider to create a query for it
    return source.Provider.CreateQuery<TSource>(methodCall);
}
```

There's only one part of this code that I don't really understand the need for, and that's the call to Expression.Quote on the predicate expression tree. I'm sure there's a good reason for it, but *this particular example* would work without it, as far as I can see. The real implementation uses it though, so dare say it's required in some way.

EDIT: Daniel's comment has made this somewhat clearer to me. Each of the arguments to Expression.Call after the MethodInfo itself is meant to be an expression which represents the argument to the method call. In our example we need an expression which represents an argument of type Expression<Func<TSource, bool>>. We already have the value, but we need to provide the layer of wrapping... just as we did with Expression.Constant in the very first expression tree I showed at the top. To wrap the expression value we've got, we use Expression.Quote. It's still not clear to me *exactly* why we can use Expression.Quote but not Expression.Constant, but at least it's clearer why we need *something*...

EDIT: I'm gradually getting there. This Stack Overflow answer from Eric Lippert has much to say on the topic. I'm still trying to get my head round it, but I'm sure when I've read Eric's answer several times, I'll get there.

We can even test that this works, by using the Queryable.AsQueryable method from the real .NET implementation. This creates an IQuerable<T> from any IEnumerable<T> using a built-in query provider. Here's the test program, where FakeQueryable is a static class containing the extension method above:

```
using System;
using System.Collections.Generic;
using System.Linq;

class Test
{
    static void Main()
    {
        List<int> list = new List<int> { 3, 5, 1 };
        IQueryable<int> source = list.AsQueryable();
        IQueryable<int> query = FakeQueryable.Where(source, x => x > 2);

        foreach (int value in query)
        {
            Console.WriteLine(value);
        }
    }
}
```

This works, printing just 3 and 5, filtering out the 1. Yay! (I'm explicitly calling

FakeQueryable.Where rather than letting extension method resolution find it, just to make things clearer.)

Um, but what's doing the actual work? We've implemented the Where clause without providing any filtering ourselves. It's really the query provider which has built an appropriate IQueryable<T> implementation. When we call GetEnumerator() implicitly in the foreach loop, the query can examine everything that's built up in the expression tree (which could contain multiple operators - it's nesting queries within queries, essentially) and work out what to do. In the case of our IQueryable<T> built from a list, it just does the filtering in-process... but if we were using LINQ to SQL, *that's* when the SQL would be generated. The provider recognizes the specific methods from Queryable, and applies filters, projections etc. That's why it was important that our demo Where method pretended that the real Queryable.Where had been called - otherwise the query provider wouldn't know what the call expression

Just to hammer the point home even further... Queryable itself neither knows nor cares what kind of data source you're using. Its job is *not* to perform any query operations itself; its job is to *record* the requested query operations in a source-agnostic manner, and let the source provider handle them when it needs to.

Immediate execution with IQueryProvider.Execute

All the operators using deferred execution in Queryable are implemented in much the same way as our demo Where method. However, that doesn't cover the situation where we need to execute the query *now*, because it has to return a value directly instead of another query.

This time I'm going to use ElementAt as the sample, simply because it's only got one overload, which makes it very easy to grab the relevant MethodInfo. The general procedure is exactly the same as building a new query, except that this time we call the provider's Execute method instead of CreateQuery.

```
public static TSource ElementAt<TSource>(this IQueryable<TSource> source, int
index)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }

    Expression sourceExpression = source.Expression;
    Expression indexExpression = Expression.Constant(index);

    MethodInfo method = typeof(Queryable).GetMethod("ElementAt");
    MethodInfo closedMethod = method.MakeGenericMethod(new Type[] {
typeof(TSource) });

    // Now we can create a *representation* of this exact method call
    Expression methodCall = Expression.Call(closedMethod, sourceExpression,
indexExpression);

    // ... and ask our query provider to execute it
    return source.Provider.Execute<TSource>(methodCall);
}
```

The type argument we provide to Execute is the desired *return* type - so for Count, we'd call Execute<int> for example. Again, it's up to the query provider to work out what the call actually means.

It's worth mentioning that both CreateQuery and Execute have generic and non-generic overloads. I haven't personally encountered a use for the non-generic ones,

but I gather they're useful for various situations in generated code, particularly if you really don't know the element type - or at least only know it dynamically, and don't want to have to use reflection to generate an appropriate generic method call.

Transparent support in source code

One of the aspects of LINQ which raises it to the "genius" status (and "slightly scary" at the same time) is that most of the time, most developers don't need to make any changes to their source code in order to use Enumerable or Queryable. Take this query expression and its translation:

```
var query = from person in family
            where person.LastName == "Skeet"
            select person.FirstName;

// Translation
var query = family.Where(person => person.LastName == "Skeet")
                  .Select(person => person.FirstName);
```

Which set of query methods will that use? It entirely depends on the compile-time type of the "family" variable. If that's a type which implements IQueryable<T>, it will use the extension methods in Queryable, the lambda expression will be converted into expression trees, and the type of "query" will be IQueryable<string>. Otherwise (and assuming the type implements IEnumerable<T> isn't some other interesting type such as ParallelEnumerable) it will use the extension methods in Enumerable, the lambda expressions will be converted into delgeates, and the type of "query" will be IEnumerable<string>.

The query expression translation part of the specification has no need to care about this, because it's simply translating into a form which uses lambda expressions - the rest of overload resolution and lambda expression conversion deals with the details.

Genius... although it does mean you need to be careful that *really* you know where your query evaluation is going to take place - you don't want to accidentally end up performing your whole query in-process having shipped the entire contents of a database across a network connection...

Conclusion

This was really a whistlestop tour of the "other" side of LINQ - and without going into any of the details of the real providers such as LINQ to SQL. However, I hope it's given you enough of a flavour for what's going on to appreciate the general design. Highlights:

- *Expression trees* are used to capture logic in a data structure which can be examined relatively easily at execution time
- Lambda expressions can be converted into expression trees as well as delegates
- IQueryable<T> and IQueryable form a sort of parallel interface hierarchy to IEnumerable<T> and IEnumerable - although the queryable forms extend the enumerable forms
- IQueryProvider enables one query to be built based on another, or executed immediately where appropriate
- Queryable provides equivalent extension methods to most of the Enumerable LINQ operators, except that it uses IQueryable<T> sources and expression trees instead of delegates
- Queryable doesn't handle the queries itself at all; it simply records what's been called and delegates the real processing to the query provider

I *think* I've now covered most of the topics I wanted to mention after finishing the actual Edulinq implementation. Next up I'll talk about some of the thorny design issues (most of which I've already mentioned, but which bear repeating) and then I'll write a brief "series conclusion" post with a list of links to all the other parts.

Back to the table of contents.

Part 44 - Aspects of Design

I promised a post on some questions of design that are raised by LINQ to Objects. I suspect that most of these have already been covered in other posts, but it may well be helpful to talk about them here too. This time I've thought about it particularly from the point of view of how other APIs can be built on some of the same design principles, and the awkward choices that LINQ has thrown up.

The power of composability and immutability

Perhaps the most important aspect of LINQ which I'd love other API designers to take on board is that of how complicated queries are constructed from lots of little building blocks. What makes it particularly elegant is that the result of applying each building block is unchanged by anything else you do to it afterwards.

LINQ doesn't *enforce* immutability of course - you can start off with a mutable list and change its content at any time, for example, or change the properties of one of the objects referenced within it, or pass in a delegate with side-effects - but *LINQ itself* won't introduce side-effects.

The Task-based Asynchronous Pattern takes a similar approach, allowing composable building blocks of tasks. I've seen this pattern in various guises over the years - if you find yourself thinking in terms of a pipeline of some kind, it may well be appropriate, especially if each state in the pipeline emits the same type as it consumes.

General immutability is a somewhat different design trait of course, but one which can make *such* a difference. The java.util.{Date,Calendar} classes are horrible, not least because they're mutable - you can never stash a value away without being concerned that it may get changed by something else. Joda Time has *some* mutable implementations, but typically the immutable classes are used in a fluent way. Of course, .NET uses value types for various core types to start with, but also makes TimeZoneInfo immutable. For genuine "values" I would highly encourage API designers to at least *strongly consider* immutable types. They're not *always* appropriate by any means, but they can be hugely useful where they fit nicely.

Extension methods on interfaces

It's no surprise that extension methods are heavily used in LINQ, given that they were effectively introduced into the language in order to enable LINQ in the first place. However, they do work particularly well with interfaces as a way of adding common

behaviour.

It also plays very nicely with the pipeline pattern above for creating pipelines in a fluent manner. Even if you just create extension methods which call a constructor to wrap/compose the previous stage in the pipeline, you can still end up with more readable code.

One *problem* with this is that you can't "override" behaviour in particular implementations or interfaces which extend the original one - which is why Enumerable.ElementAt() has to detect that a sequence is actually a list, for example. If interfaces allowed method implementations, this wouldn't be as much of a problem in the situation where you're in control of the interface - I wouldn't be at all surprised to find that as a feature of C#'s successor.

The lack of extension properties is also a bit of a handicap in some places, although not as many as one might expect at first glance. For example, even if we *could* have made Enumerable.Count() a property, would it have been a good idea to do so? Properties give a natural expectation of speed, and Count() is usually an O(n) operation.

Delegates for custom behaviour

In .NET 1.0 and 1.1, most developers used delegates for two purposes:

- Handling events in UIs
- Passing around behaviour to be executed in a different thread (either via Control.Invoke, or new Thread(ThreadStart), or ThreadPool.QueueUserWorkItem).

.NET 2.0 increased the range of uses of delegates somewhat, particularly with List.ConvertAll and the ability to create delegates relatively easily using anonymous methods.

However, LINQ *really* brought them into the mainstream. If you're building an API which benefits from *small* pieces of custom behaviour, delegates can be a real boon. More complicated behaviour is still often best represented via an interface, and *sometimes* it's worth having both interface and delegate representations, like Comparison<T> and IComparable<T>. It's generally easy to convert between the two - especially if you use a method group conversion from an interface implementation's method to the delegate type.

Laziness

One aspect of LINQ which is both a blessing and a curse is its laziness, both in terms of deferred execution (not reading from the input sequence *at all* until the result sequence is read) and in terms of streaming the data (only reading as much information from the input sequence as is required to answer the immediate needs of the caller).

This is great in various ways, particularly as it means you can build a complex query and use it multiple times, sometimes as a basis for other queries, knowing that it won't actually do anything until you ask for real results. It also means that you can iterate over huge data sets, so long as you're careful.

On the other hand, it leads to subtle issues over when code is actually executing, makes debugging harder to understand, makes it easier to accidentally change the values of captured variables between the point at which you create the query and the point at which you execute it, and basically messes with your head. This is probably the aspect of LINQ which confuses newbies more than any other.

I'm not saying it was the wrong decision for LINQ - but I *would* caution API designers to think carefully before introducing laziness, and to document it *really* thoroughly. Likewise if your API might end up returning a result which is "gradually evaluated" (streaming data etc), this should be made clear.

When names collide: options for consistency

Just in case you've forgotten, this is irritation with the meaning of source.Contains(element). In order to check whether a sequence contains an element or not, there has to be some idea of equality - for example, if you're trying to find one string in a sequence of strings, are you trying to match in a case sensitive manner or not?

There's an overload for Enumerable.Contains which allows you to specify the equality comparer to use, but the question is what should happen when you let the implementation pick the comparer.

For *every* other method in Enumerable, the default equality comparer for the sequence type - i.e. EqualityComparer<TSource>.Default - is picked. That sounds like source.Contains(element) should use the element type's default comparer too, right? Well, in some cases that's what will happen... but not if the source implements ICollection<T>, which has its own Contains method which doesn't take an equality comparer. If that's the case, LINQ to Objects delegates to the collection's Contains method.

So, we have three kinds of consistency here:

- Consistency of compile-time type: it would be nice if the behaviour of source.Contains(element) was the same whether "source" is of type IEnumerable<T> and ICollection<T>
- Consistency of API: it would be nice if Contains behaved the same way as other methods which have overloads with and without equality comparers
- Consistency of model: if you consider "source" to be just a sequence of elements, it shouldn't affect the *result* (not just the speed) if the object actually implements ICollection<T>

I should point out that this will *only* be a problem if the collection uses a different notion of equality to the default equality comparer for the type. The most obvious example of this is if you have a HashSet<string> which uses a case-insensitive equality comparer. But it's still a valid concern.

So what should the API designer do in this kind of case? Admittedly LINQ to Objects is already in a slightly unusual position as it's based on an existing interface with known and *very common* interfaces extending that core one... it's less likely to come up with other APIs. However, I think it *might* be enough of a smell to suggest that changing the name of the method to "ContainsElement" or something similar would be worthwhile. It's unfortunate that "Contains" really *is* the obvious choice...

This issue raises another aspect of API decision I've considered in the past... if there's a common way of doing something in the framework you're building on top of, but you consider it to be broken, should you abide by that breakage for the sake of familiarity and consistency, or should you strive to be as "clean" as possible? I think it needs to be considered on a case-by-case basis, but I *suspect* I would usually come down on the side of cleanliness.

Documentation details

Almost all APIs are badly documented - it's a fact of life, even with some of the best APIs I've worked with. I doubt that Noda Time will be a shining example either. However, at the risk of being hypocritical I'll say that documentation is worth spending significant thought on. Not just the time taken to document your code - but the time taken to consider what you want to guarantee, what should be left unstated, and what should be *explicitly* left open.

For example, there's no indication in the documentation of Cast that it will *sometimes* return the original source value, nor in its companion OfType method that that will *never* return the original source reference. This might be important to someone - why not state it? It's possible to state the *possibility* without saying what cases it applies to of course, leaving some wiggle room in the future. You might consider some of the optimizations in the same way - when should an optimization be documented and

when should it be implicit? Sometimes it can make a difference beyond just performance, even if only in "odd" situations (such as a predicate throwing an exception).

If you're used to defensive coding with Code Contracts, it's much the same type of decision - and again, it's similar to deciding whether a method should return IEnumerable<T>, IList<T> or List<T>. There's a balance between caller convenience, design cleanliness (where you only want to *emphasize* one interface aspect, even if it also *happens* to always return a particular type), and room for the implementation to change in the future.

Another example of considering the level of detail to document is when it comes to how input sequences are used in LINQ to Objects. What does it mean to say "this method uses deferred execution" *exactly*? If I call GetEnumerator() eagerly but defer the call to MoveNext(), is that still "deferred execution"? Should the documentation state when a sequence is buffered and when it's streamed? Should it guarantee the order of the result sequence when the natural implementation makes that order easy to describe (e.g. for Distinct)? In this series I've tried to be as clear as possible about what actually happens - but that's not to say that in some cases, the documentation wasn't left *deliberately* ambiguous.

Conclusion

There are many other design considerations that I haven't gone into here - particularly optimization, which I've already covered twice, probably saying everything I wanted to say here anyway.

I may add a few more bits to this post over time... but aside from that, I think I'm fundamentally *done*. I'll write one more conclusion post, then declare Edulinq closed...

Back to the table of contents.

Part 45 - Conclusion

(Table of contents omitted from HTML archive; see the separate table of contents page.

When I started this series, I hadn't realised quite how much there would be to write about. The main thrust was going to be that the implementation of LINQ is simple, and it's the design that's clever. As it happened, pretty much every operator ended up raising *some* interesting issue or other. However, hopefully the series has still "immersed" you in LINQ to Objects to some extent, and clarified how it all hangs together. It would be gratifying to think that the description at the start of each post may end up being used as a sort of "unofficial alternative documentation" with some more details than MSDN provides, but we'll see whether than happens over time.

A number of people have asked me whether there'll be an ebook version of this series, and the answer is currently "I don't know." I have a few plans afoot, but I can't tell where they'll lead yet. Suffice to say I like the idea, and I'm looking at some options.

Anyway, thank you for reading as much of the series as you have, and I hope you've enjoyed it as much as I have.

Just in case you're wondering, I'll probably go back to posting about C# 5's async support pretty soon...

Back to the table of contents.