The Beauty of Closures

Some time soon, I want to write about the various Java 7 closures proposals on my blog. However, when I started writing that post I found it was difficult to start off without an introduction to closures. As time went by, the introduction became so long that I feared I'd lose most of my readers before I got to the Java 7 bit. As chapter 5 in the book is largely about anonymous methods and how they provide closures to C#, it seemed appropriate to write this article here.

Most articles about closures are written in terms of functional languages, as they tend to support them best. However, that's also precisely why it's useful to have an article written about how they appear more traditional OO languages. Chances are if you're writing in a functional language, you know about them already. This article will talk about C# (versions 1, 2 and 3) and Java (before version 7).

What are closures?

To put it very simply, closures allow you to encapsulate some behaviour, pass it around like any other object, and still have access to the context in which they were first declared. This allows you to separate out control structures, logical operators etc from the details of how they're going to be used. The ability to access the original context is what separates closures from normal objects, although closure implementations typically achieve this using normal objects and compiler trickery.

It's easiest to look at a lot of the benefits (and implementations) of closures with an example. I'll use a single example for most of the rest of this article. I'll show the code in Java and C# (of different versions) to illustrate different approaches. All the code is also available for download so you can tinker with it.

Example situation: filtering a list

It's reasonably common to want to filter a list by some criterion. This is quite easy to do "inline" by just creating a new list, iterating over the original list and adding the appropriate elements to the new list. It only takes a few lines of code, but it's still nice to hide that logic away in one place. The difficult bit is encapsulating which items to include. This is where closures come in.

Although I've used the word "filter" in the description, it's somewhat ambiguous between filtering items *in* the new list and filtering things *out*. Does an "even number filter" keep or reject even numbers, for instance? We'll use a different bit of terminology - a *predicate*. A predicate is simply something which matches or doesn't match a given item. Our example will produce a new list containing every element of the original list which matches the given predicate.

In C# the natural way of representing a predicate is as a delegate, and indeed .NET 2.0 contains a `Predicate<T>` type. (Aside: for some reason LINQ prefers `Func<T,bool>`; I'm not sure why, given that it's less descriptive. The two are functionally equivalent.) In Java there's no such thing as a delegate, so we'll use an interface with a single method. Of

course we *could* use an interface in C# as well, but it would be significantly messier and wouldn't let us use anonymous methods and lambda expressions - which are precisely the features which implement closures in C#. Here are the interface/delegate for reference:

```
// Declaration for System.Predicate<T>
public delegate bool Predicate<T>(T obj)
```

```
// Predicate.java
public interface Predicate<T>
{
    boolean match(T item);
}
```

The code used to filter the list is very straightforward in both languages. I should point out at this stage that I'm going to steer clear of extension methods in C# just to make the example simpler - but anyone who has used LINQ should be reminded of the `Where` extension method. (There are differences in terms of deferred execution, but I'll avoid those for the moment.)

```
// In ListUtil.cs
static class ListUtil
{
    public static IList<T> Filter<T>(IList<T> source, Predicate<T> predicate)
    {
        List<T> ret = new List<T>();
        foreach (T item in source)
        {
            if (predicate(item))
            {
                ret.Add(item);
            }
        }
        return ret;
    }
}
```

```
// In ListUtil.java
public class ListUtil
{
    public static <T> List<T> filter(List<T> source, Predicate<T> predicate)
    {
        ArrayList<T> ret = new ArrayList<T>();
        for (T item : source)
        {
            if (predicate.match(item))
            {
                ret.add(item);
            }
        }
        return ret;
    }
}
```

(In both languages I've included a `Dump` method in the same class which just writes out the given list to the console.)

Now that we've defined our filtering method, we need to call it. In order to demonstrate

the importance of closures, we'll start with a simple case which can be solved without them, and then move on to something harder.

Filter 1: Matching short strings (length fixed)

Our sample situation is going to be really basic, but I hope you'll be able to see the significance anyway. We'll take a list of strings, and then produce another list which contains only the "short" strings from the original list. Building a list is clearly simple - it's creating the predicate which is the tricky bit.

In C# 1, we have to have a method to represent the logic of our predicate. The delegate instance is then created by specifying the name of the method. (Of course this code isn't actually valid C# 1 due to the use of generics, but concentrate on how the delegate instance is being created - that's the important bit.)

```
// In Example1a.cs
static void Main()
{
    Predicate<string> predicate = new Predicate<string>(MatchFourLettersOrFewer);
    IList<string> shortWords = ListUtil.Filter(SampleData.Words, predicate);
    ListUtil.Dump(shortWords);
}

static bool MatchFourLettersOrFewer(string item)
{
    return item.Length <= 4;
}
```

In C# 2, we have three options. We can use exactly the same code as before, or we can simplify it *very slightly* using the new method group conversions available, or we can use an *anonymous method* to specify the predicate's logic "inline". The enhanced method group conversion option isn't worth spending very much time on - it's just a case of changing `new Predicate<string>(MatchFourLettersOrFewer)` to `MatchFourLettersOrFewer`. It's available in the downloadable code though (in `Example1b.cs`). The anonymous method option is much more interesting:

```
static void Main()
{
    Predicate<string> predicate = delegate(string item)
        {
            return item.Length <= 4;
        };
    IList<string> shortWords = ListUtil.Filter(SampleData.Words, predicate);
    ListUtil.Dump(shortWords);
}
```

We no longer have an extraneous method, and the behaviour of the predicate is obvious at the point of use. Good stuff. How does this work behind the scenes? Well, if you use ildasm or Reflector to look at the generated code, you'll see that it's pretty much the same as the previous example: the compiler has just done some of the work for us. We'll see later on how it's capable of doing a lot more...

In C# 3 you have all the same options as before, but also *lambda expressions*. For the purposes of this article, lambda expressions are really just anonymous methods in a concise form. (The big difference between the two when it comes to LINQ is that lambda expressions can be converted to expression trees, but that's irrelevant here.) The code

looks like this when using a lambda expression:

```
static void Main()
{
    Predicate<string> predicate = item => item.Length <= 4;
    IList<string> shortWords = ListUtil.Filter(SampleData.Words, predicate);
    ListUtil.Dump(shortWords);
}
```

Ignore the fact that by using the `<=` it looks like we've got a big arrow pointing at `item.Length` - I kept it that way for consistency, but it could equally have been written as `Predicate<string> predicate = item => item.Length < 5;`

In Java we don't have to create a delegate - we have to implement an interface. The *simplest* way is to create a new class to implement the interface, like this:

```
// In FourLetterPredicate.java
public class FourLetterPredicate implements Predicate<String>
{
    public boolean match(String item)
    {
        return item.length() <= 4;
    }
}

// In Example1a.java
public static void main(String[] args)
{
    Predicate<String> predicate = new FourLetterPredicate();
    List<String> shortWords = ListUtil.filter(SampleData.WORDS, predicate);
    ListUtil.dump(shortWords);
}
```

That doesn't use any fancy language features, but it does involve a whole separate class just to express one small piece of logic. Following Java conventions, the class is likely to be in a different file, making it harder to read the code which is using it. We could make it a *nested* class instead, but the logic is still away from the code which uses it - it's a more verbose version of the C# 1 solution, effectively. (Again, I won't show the nested class version here, but it's in the downloadable code as `Example1b.java`.) Java *does* allow the code to be expressed inline, however, using *anonymous classes*. Here's the code in all its glory:

```
// In Example 1c.java
public static void main(String[] args)
{
    Predicate<String> predicate = new Predicate<String>()
    {
        public boolean match(String item)
        {
            return item.length() <= 4;
        }
    };

    List<String> shortWords = ListUtil.filter(SampleData.WORDS, predicate);
    ListUtil.dump(shortWords);
}
```

As you can see, there's a lot of syntactic noise here compared with the C# 2 and 3

solutions, but at least the code is visible in the right place. This is Java's current support for closures... which leads us nicely into the second example.

Filter 2: Matching short strings (variable length)

So far our predicate hasn't needed any context - the length is hard-coded, and the string to check is passed to it as a parameter. Let's change the situation so that the user can specify the maximum length of strings to allow.

First we'll go back to C# 1. That doesn't have any real closure support - there's no simple place to store the piece of information we need. Yes, we could just use a variable in the current context of the method (e.g. a static variable in the main class from our first example) but this is clearly not a nice solution - for one thing, it immediately removes thread safety. The answer is to separate out the required state from the current context, by creating a new class. At this point it looks very much like the original Java code, just with a delegate instead of an interface:

```
// In VariableLengthMatcher.cs
public class VariableLengthMatcher
{
    int maxLength;

    public VariableLengthMatcher(int maxLength)
    {
        this.maxLength = maxLength;
    }

    /// <summary>
    /// Method used as the action of the delegate
    /// </summary>
    public bool Match(string item)
    {
        return item.Length <= maxLength;
    }
}

// In Example2a.cs
static void Main()
{
    Console.Write("Maximum length of string to include? ");
    int maxLength = int.Parse(Console.ReadLine());

    VariableLengthMatcher matcher = new VariableLengthMatcher(maxLength);
    Predicate<string> predicate = matcher.Match;
    IList<string> shortWords = ListUtil.Filter(SampleData.Words, predicate);
    ListUtil.Dump(shortWords);
}
```

The change to the code for both C# 2 and C# 3 is simpler: we just replace the hard-coded limit with the parameter in both cases. Don't worry about exactly how this works just yet - we'll examine that when we've seen the Java code in a minute.

```
// In Example2b.cs (C# 2)
static void Main()
{
    Console.Write("Maximum length of string to include? ");
    int maxLength = int.Parse(Console.ReadLine());

    Predicate<string> predicate = delegate(string item)
```

```
    {
        return item.Length <= maxLength;
    };
    IList<string> shortWords = ListUtil.Filter(SampleData.Words, predicate);
    ListUtil.Dump(shortWords);
}
```

```
// In Example2c.cs (C# 3)
static void Main()
{
    Console.Write("Maximum length of string to include? ");
    int maxLength = int.Parse(Console.ReadLine());

    Predicate<string> predicate = item => item.Length <= maxLength;
    IList<string> shortWords = ListUtil.Filter(SampleData.Words, predicate);
    ListUtil.Dump(shortWords);
}
```

The change to the Java code (the version using anonymous classes) is similar, but with one little twist - we have to make the parameter `final`. It sounds odd, but there's method in Java's madness. Let's look at the code before working out what it's doing:

```
// In Example2a.java
public static void main(String[] args) throws IOException
{
    System.out.print("Maximum length of string to include? ");
    BufferedReader console = new BufferedReader(new InputStreamReader(System.in));
    final int maxLength = Integer.parseInt(console.readLine());

    Predicate<String> predicate = new Predicate<String>()
    {
        public boolean match(String item)
        {
            return item.length() <= maxLength;
        }
    };

    List<String> shortWords = ListUtil.filter(SampleData.WORDS, predicate);
    ListUtil.dump(shortWords);
}
```

So, what's the difference between the Java and the C# code? *In Java, the **value** of the variable has been captured by the anonymous class. In C#, the **variable itself** has been captured by the delegate.* To prove that C# captures the variable, let's change the C# 3 code to change the value of the parameter after the list has been filtered once, and then filter it again:

```
// In Example2d.cs
static void Main()
{
    Console.Write("Maximum length of string to include? ");
    int maxLength = int.Parse(Console.ReadLine());

    Predicate<string> predicate = item => item.Length <= maxLength;
    IList<string> shortWords = ListUtil.Filter(SampleData.Words, predicate);
    ListUtil.Dump(shortWords);

    Console.WriteLine("Now for words with <= 5 letters:");
    maxLength = 5;
    shortWords = ListUtil.Filter(SampleData.Words, predicate);
```

```
    ListUtil.Dump(shortWords);
}
```

Note that we're *only* changing the value of the local variable. We're not recreating the delegate instance, or anything like that. The delegate instance has access to the local variable, so it can see that it's changed. Let's go one step further, and make the predicate itself change the value of the variable:

```
// In Example2e.cs
static void Main()
{
    int maxLength = 0;

    Predicate<string> predicate = item => { maxLength++; return item.Length <= maxLength; };
    IList<string> shortWords = ListUtil.Filter(SampleData.Words, predicate);
    ListUtil.Dump(shortWords);
}
```

I'm not going to go into the details of how all this is achieved - read chapter 5 of C# in Depth for the gory stuff. Just expect to have some of your notions of what "local variable" means turned upside down.

Having seen how C# reacts to changes in captured variables, what happens in Java? Well, it's a pretty simple answer: you can't change the value of a captured variable. It *has* to be `final`, so the question is moot. However, if somehow you *could* change the value of the variable, you'd find that the predicate didn't respond to it. The values of captured variables are copied when the predicate is created, and stored in the instance of the anonymous class. For reference variables, don't forget that the value of the variable is just the *reference*, not the current state of the object. For example, if you capture a `StringBuilder` and then append to it, those changes *will* be seen in the anonymous class.

Comparing capture strategies: complexity vs power

Clearly the Java scheme is more restrictive, but it does make life significantly simpler, too. Local variables behave in the same way they've always done, and in many cases the code is easier to understand, too. For example, look at the following code, using the Java `Runnable` interface and the .NET `Action` delegate - both of which represent actions taking no parameters and returning no value. First let's see the C# code:

```
// In Example3a.cs
static void Main()
{
    // First build a list of actions
    List<Action> actions = new List<Action>();
    for (int counter = 0; counter < 10; counter++)
    {
        actions.Add(() => Console.WriteLine(counter));
    }

    // Then execute them
    foreach (Action action in actions)
    {
        action();
    }
}
```

What's the output? Well, we've only actually declared a single `counter` variable - so that same `counter` variable is captured by all the `Action` instances. The result is the number 10 being printed on every line. To "fix" the code to make it display the output most people would expect (i.e. 0 to 9) we need to introduce an extra variable inside the loop:

```
// In Example3b.cs
static void Main()
{
    // First build a list of actions
    List<Action> actions = new List<Action>();
    for (int counter = 0; counter < 10; counter++)
    {
        int copy = counter;
        actions.Add(() => Console.WriteLine(copy));
    }

    // Then execute them
    foreach (Action action in actions)
    {
        action();
    }
}
```

Each time we go through the loop we're said to get a different *instance* of the `copy` variable - each `Action` captures a different variable. This makes perfect sense if you look at what the compiler's actually doing behind the scenes, but initially it flies in the face of the intuition of most developers (including me).

Java forbids the first version entirely - you can't capture the `counter` variable at all, because it's not final. To use a final variable, you end up with code like this, which is very similar to the C# code:

```
// In Example3a.java
public static void main(String[] args)
{
    // First build a list of actions
    List<Runnable> actions = new ArrayList<Runnable>();
    for (int counter=0; counter < 10; counter++)
    {
        final int copy = counter;
        actions.add(new Runnable()
        {
            public void run()
            {
                System.out.println(copy);
            }
        });
    }

    // Then execute them
    for (Runnable action : actions)
    {
        action.run();
    }
}
```

The meaning is reasonably clear with the "captured value" semantics. The resulting code is still less pleasant to look at than the C# due to the wordier syntax, but Java forces the correct code to be written as the only option. The downside is that when you *want* the behaviour of the original C# code (which is certainly the case on occasion) it's

cumbersome to achieve in Java. (You can have a single-element array, and capture a reference to the array, then change the value of the element when you want to, but that's a nasty kludge).

What's the big deal?

In the example, we've only seen a little bit of benefit in using a closure. Sure, we've separated out the control structure side from the logic required in the filtering itself, but that's not made the code *that* much simpler on its own. This is a familiar situation - a new feature often looks less-than-impressive when used in simplistic examples. The benefit that closures often bring, however, is *composability*. If that sounds like a bit of a stretch, I agree - and that's part of the problem. Once you're familiar with closures and possibly a little addicted to them, the connection seems very obvious. Until that time, it seems obscure.

Closures don't inherently provide composability. All they do is make it simpler to implement delegates (or single-method interfaces - I'll stick to using the term delegates for simplicity). Without some support for closures, it's easier to write a small loop than it is to call another method to do the looping, providing a delegate for some piece of the logic involved. Even with "just add a method to an existing class" support for delegates, you still end up losing the locality of the logic, *and* you often need more contextual information than it's easy to provide.

So, closures make it simpler to create delegates. That means it's more worthwhile to design APIs which *use* delegates. (I don't believe it's a coincidence that delegates were almost solely used for starting threads and handling events in .NET 1.1.) Once you start thinking in terms of delegates, the ways to combine them become obvious. For instance, it's trivial to create a `Predicate<T>` which takes two other predicates, and represents either the logical AND or OR of them (or other boolean operations, of course).

A different sort of combination comes when you feed the result of one delegate into another, or curry one delegate to create a new one. All sorts of options become available when you start thinking of the logic as just another type of data.

The use of composition doesn't end there though - the whole of LINQ is built on it. The filter we built using lists is just one example of how one sequence of data can be transformed into another. Other operations include ordering, grouping, combining with another sequence, and projecting. Writing each of those operations out longhand hasn't been too painful historically, but the complexity soon mounts up by the time your "data pipeline" consists of more than a few transformations, In addition, with the deferred execution and data streaming provided by LINQ to Objects, you incur significantly less memory overhead than with in the straightforward implementation of just running one transformation when the other has finished. The complexity isn't taken out of the equation by the individual transformations being particularly clever - it's removed by the ability to express small snippets of logic inline with closures, and the ability to combine operations with a well-designed API.

Conclusion

Closures are a little underwhelming to start with. Sure, they let you implement an interface or create a delegate instance pretty simply (depending on language). Their power only becomes evident when you use them with libraries which take advantage of them, allowing you to express custom behaviour at just the right place. When the same

libraries also allow you to compose several simple steps in a natural manner to implement significant behaviour, you get a whole which is only as complex as the sum of its parts - instead of as complex as the *product* of its parts. While I don't quite buy into the idea of composability as the silver bullet against complexity which some would advocate, it's certainly a powerful technique, and one which is applicable in many more situations due to closures.

One of the key features of lambda expressions is the brevity. When you compare the Java code from earlier with the C# code, Java looks extremely clumsy and heavyweight. This is one of the issues which the various proposals for closures in Java aim to address. I will give my perspective on these different proposals in a blog post in the not-too-distant future.