Higher Order Programming in C# with LINQ

CIS399-005 Code Appendix

Jeff Vaughan

April 15, 2009

1 Introduction

This class demonstrates a *higher-order programming* style of writing code to access data structures or other collections of values. They key idea is that we avoid using iteration to explicitly, and instead define special functions that manipulate collections.

To begin, consider the `IEnumerator<T>` interface defined in `System.Collections.Generic` and listed below.

```
interface IEnumerator<T> : IDisposable, IEnumerator {

  // Gets the current element in the collection.
  // Requires that the previous call to MoveNext() returned true.
  T Current { get; }

  // Un-generic overload of current.  Inherited from IEnumerator.
  object Current { get; }

  // Advances the enumerator to the next element of the collection.
  // MoveNext() returns false iff there is no next element.
  bool MoveNext()

  // Sets the enumerator to its initial position, which is before the first
  // element in the collection.
  void Reset()

  // Inherited from IDisposable.  We will ignore this.
  void Dispose();
}
```

Using this interface, we can access a data structure or other sequence without worrying about its underlying representation.

2 An enumerator defining the sequence 1...10

```
using System;
using System.Collections.Generic;

class TenEnumerator : IEnumerator<int>
{
    // There's some ambinguity about the correct behavoir of
    // Current.  MSDN says the get should throw an exception if
    // MoveNext() last returned false.  Section 10.14.4.2 of the C#
    // spec says the operation's behavoir is undefined.
    public int Current
    {
        get; private set;
    }

    object System.Collections.IEnumerator.Current
    {
        get { return Current; }
    }

    public void Reset()
    {
        Current = 0;
    }

    public void Dispose() { }

    public bool MoveNext(){
        Current++;
        return (Current <= 10);
    }

    public TenEnumerator()
    {
        Reset();
    }
}

class Printer
{

    static void Main()
    {
        var e = new TenEnumerator();

        while (e.MoveNext())
        {
            Console.WriteLine(e.Current);
        }
    }
}
```

3 An enumerator for a sequence with settable range

```
using System;
using System.Collections.Generic;

class SeqEnumerator : IEnumerator<int>
{
    int min;
    int max;

    public int Current
    {
        get; private set;
    }

    object System.Collections.IEnumerator.Current
    {
        get { return Current; }
    }

    public void Reset()
    {
        Current = min-1;
    }

    public void Dispose() { }

    public bool MoveNext(){
        Current++;
        return (Current <= max);
    }

    private SeqEnumerator() {}

    public SeqEnumerator(int min, int max)
    {
        this.min = min;
        this.max = max;
        Reset();
    }
}

class Program
{
    static void Main(string[] args)
    {
        IEnumerator<int> e = new SeqEnumerator( -3, 4);

        while (e.MoveNext())
        {
            Console.WriteLine(e.Current);
        }

        Console.ReadKey();
    }
}
```

4 Collection are objects that can be accessed with enumerators.

Collections must implement the `IEnumerable<T>` interface. This contains two members: one of which returns an `IEnumerator`, the other an `IEnumerator<T>`.

```
using System.Collections.Generic;
using System;

class SeqEnumerator : IEnumerator<int>
{
    int min;
    int max;

    public int Current
    {
        get; private set;
    }

    object System.Collections.IEnumerator.Current
    {
        get { return Current; }
    }

    public void Reset()
    {
        Current = min-1;
    }

    public void Dispose() { }

    public bool MoveNext(){
        Current++;
        return (Current <= max);
    }

    private SeqEnumerator() {}

    public SeqEnumerator(int min, int max)
    {
        this.min = min;
        this.max = max;
        Reset();
    }
}

//NumericalRange represents a collection of integers.
class NumericalRange : IEnumerable<int>
{
    private int first;
    private int length;

    public NumericalRange(int first, int length)
    {
        this.first = first;
        this.length = length;
    }
```

```
        public IEnumerator<int> GetEnumerator()
        {
            return new SeqEnumerator(first, first+length);
        }

        System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var nr = new NumericalRange( 4, 11);
            IEnumerator<int> e = nr.GetEnumerator();

            while (e.MoveNext())
            {
                Console.WriteLine(e.Current);
            }

            Console.ReadKey();
        }
    }
```

5 Introducing yield

Enumerators and Enumerable can be built automatically using *iterators* (also called *co-routines*). An iterator is method which *yields* several answers in sequence.

The example below shows how an iterator can be used to automatically define and Enumerator.

```
using System.Collections;
using System.Collections.Generic;
using System;

class NumericalRange : IEnumerable<int>
{
    private int first;
    private int length;

    public NumericalRange(int first, int length)
    {
        this.first = first;
        this.length = length;
    }

    public IEnumerator<int> GetEnumerator()
    {
        for (int i = first; i < first + length; i++)
        {
            yield return i;
        }

    }

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

class Program
{
    static void Main(string[] args)
    {
        var nr = new NumericalRange( 4, 11);
        IEnumerator<int> e = nr.GetEnumerator();

        while (e.MoveNext())
        {
            Console.WriteLine(e.Current);
        }

        Console.ReadKey();
    }
}
```

6 Foreach loops are defined over enumerators.

```
using System.Collections;
using System.Collections.Generic;
using System;

class NumericalRange : IEnumerable<int>
{
    private int first;
    private int length;

    public NumericalRange(int first, int length)
    {
        this.first = first;
        this.length = length;
    }

    public IEnumerator<int> GetEnumerator()
    {
        for (int i = first; i < first + length; i++)
        {
            yield return i;
        }

    }

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

class Program
{
    static void Main(string[] args)
    {
        var nr = new NumericalRange( 4, 11);

        foreach (int i in nr)
        {
            Console.WriteLine(i);
        }

        Console.ReadKey();
    }
}
```

7 Defining a filter transformation by iterating

The filter operation transforms an input collection (instance of `IEnumerable<T>` with many elements into an enumerator with fewer elements. This is done without modifying the input collection. Note that the following example does this by using an iterator block to define the new collection.

```
using System.Collections;
using System.Collections.Generic;
using System;

class NumericalRange : IEnumerable<int>
{
    private int first;
    private int length;

    public NumericalRange(int first, int length)
    {
        this.first = first;
        this.length = length;
    }

    public IEnumerator<int> GetEnumerator()
    {
        for (int i = first; i < first + length; i++)
        {
            yield return i;
        }

    }

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

static class Helpers
{
    public static bool IsOdd(int x) { return (x % 2 == 1); }

    // Note that here were are defining a collection (an IEnumerable<int>)
    // using an iterator, not just an enumerator.
    public static IEnumerable<int> FilterOdds(IEnumerable<int> input)
    {
        foreach (int i in input)
        {
            if (IsOdd(i))
                yield return i;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
```

```
            var nr = new NumericalRange( 4, 11);
            var nrr = Helpers.FilterOdds(nr);

            foreach (int i in nrr)
            {
                Console.WriteLine(i);
            }

            Console.ReadKey();
        }
    }
```

8 Generalizing filter as a higher order function

We can write a generalized `Filter` that takes a predicate as an argument, and selects all elements that satisfy the predicate. Using the generalized `Filter` we can select all sorts of different things (even numbers, big numbers, etc.) without writing any more loops!

`Filter` is called a higher-order method because it takes a function (delegate) as an argument. The terminology is from functional programming languages, such as Scheme or F#.

```
static class Helpers
{
    public static bool IsOdd(int x) { return (x % 2 == 1); }
    public static bool IsEven(int x) { return (!IsOdd(x)); }
    public static bool IsBig(int x) {return (x >= 8); }

    public delegate bool IntPredicate(int x);

    public static IEnumerable<int> Filter(IEnumerable<int> input,
                                          IntPredicate f)
    {
        foreach (int i in input)
        {
            if (f(i))
                yield return i;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        var nr = new NumericalRange( 4, 11);
        var nrr = Helpers.Filter(nr, Helpers.IsEven);
        var nrrr = Helpers.Filter(nrr, Helpers.IsBig);

        foreach (int i in nrrr)
        {
            Console.WriteLine(i);
        }

        Console.ReadKey();
    }
}
```

9 Calling filter with an anonymous lambda

```
    static void Main(string[] args)
    {
        var nr = new NumericalRange( 4, 11);
        var nrr = Helpers.Filter(nr, Helpers.IsEven);
        var nrrr = Helpers.Filter(nrr, Helpers.IsBig);
        var nrrrr = Helpers.Filter(nrrr, x => (x < 12));

        foreach (int i in nrrrr)
        {
            Console.WriteLine(i);
        }

        Console.ReadKey();
    }
```

10 Map: another higher-order function

```
static class Helpers
{
    public static bool IsOdd(int x) { return (x % 2 == 1); }
    public static bool IsEven(int x) { return (!IsOdd(x)); }
    public static bool IsBig(int x) {return (x >= 8); }

    public delegate bool IntPredicate(int x);
    public delegate int IntOperator(int x);

    public static IEnumerable<int> Filter(IEnumerable<int> input,
                                          IntPredicate f)
    {
        foreach (int i in input)
        {
            if (f(i))
                yield return i;
        }
    }

    public static IEnumerable<int> Map(IEnumerable<int> input,
                               IntOperator f)
    {
        foreach (int i in input)
            yield return f(i);
    }
}

class Program
{
    static void Main(string[] args)
    {
        var nr = new NumericalRange( 4, 11);
        var nrr = Helpers.Filter(nr, Helpers.IsEven);
        var nrrr = Helpers.Filter(nrr, Helpers.IsBig);
        var nrrrr = Helpers.Map(nrrr, x => x * x);

        foreach (int i in nrrrr)
        {
            Console.WriteLine(i);
        }

        Console.ReadKey();
    }
}
```

11 Generalizing using generics

Nothing about `Map` of `Filter` really depends on on the fact the underlying collection contains integers. We can generalize these methods to be generic over the collection element type!

```
static class Helpers
{
    public delegate S Function<T,S>(T x);

    public static IEnumerable<T> Filter<T>(IEnumerable<T> input,
                                           Function<T,bool> f)
    {
        foreach (T i in input)
        {
            if (f(i))
                yield return i;
        }
    }

    public static IEnumerable<S> Map<T,S>(IEnumerable<T> input,
                                          Function<T,S> f)
    {
        foreach (T i in input)
            yield return f(i);
    }
}

class Program
{
    static void Main(string[] args)
    {
        var nr = new NumericalRange( 4, 11);
        var nrr = Helpers.Filter(nr, x => x > 10);
        var nrrr = Helpers.Map(nrr, x => x * x);

        foreach (int i in nrrr)
        {
            Console.WriteLine(i);
        }

        Console.ReadKey();
    }
}
```

12 Rewriting with extension methods

Extension methods provide syntax a syntax for "adding" methods to a class from outside the class's definition. This is just syntactic sugar; extension methods cannot access private fields.

```
static class Helpers
{
    public delegate S Function<T,S>(T x);

    // The keyword this (attached to the first argument)
    // indicates that Filter is an extension method for interface
    // IEnumerable<T>.
    public static IEnumerable<T> Filter<T>(this IEnumerable<T> input,
                                           Function<T,bool> f)
    {
        foreach (T i in input)
        {
            if (f(i))
                yield return i;
        }
    }

    public static IEnumerable<S> Map<T,S>(this IEnumerable<T> input,
                                          Function<T,S> f)
    {
        foreach (T i in input)
            yield return f(i);
    }
}

class Program
{
    static void Main(string[] args)
    {
        var nr = new NumericalRange( 4, 11).Filter(x => x > 10)
                                           .Map(x => x * x);

        foreach (int i in nr)
        {
            Console.WriteLine(i);
        }

        Console.ReadKey();
    }
}
```

13 Replacing custom code with LINQ library calls

`System.Linq.Where` and `System.Linq.Select` do the same thing as our `Map` and `Filter`.

```
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System;

class NumericalRange : IEnumerable<int>

...
}

// Deleted class Helper, as Linq covers its functionality.

class Program
{
    static void Main(string[] args)
    {
        var nr = new NumericalRange( 4, 11).Where(x => x > 10)
                                           .Select(x => x * x);

        foreach (int i in nr)
        {
            Console.WriteLine(i);
        }

        Console.ReadKey();
    }
}
```

14 Using special LINQ syntax

C# provides special, SQL-inspired syntax for some of these methods, including `Where` and `Select`.

```
using System.Collections;
using System.Collections.Generic;
using System;
using System.Linq;

class NumericalRange : IEnumerable<int>
{
    private int first;
    private int length;

    public NumericalRange(int first, int length)
    {
        this.first = first;
        this.length = length;
    }

    public IEnumerator<int> GetEnumerator()
    {
        for (int i = first; i < first + length; i++)
        {
            yield return i;
        }

    }

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

class Program
{
    static void Main(string[] args)
    {
        var nr = from x in (new NumericalRange(4, 11))
                 where x > 10
                 select x*x;

        foreach (int i in nr)
        {
            Console.WriteLine(i);
        }

        Console.ReadKey();
    }
}
```