Functional C#

- LINQ
- Closures
- Lambda expressions
- Anonymous functions
- Parameterized classes and functions
- Currying

LINQ

(http://en.wikipedia.org/wiki/Language_Integrated_Query)

LINQ defines a set of method names (called standard query operators, or standard sequence operators), along with translation rules from so-called query expressions to expressions using these method names, lambda expressions and anonymous types. These can, for example, be used to project and filter data into arrays, enumerable classes, XML (LINQ to XML), relational databases, and third party data sources. Other uses, which use query expressions as a general framework for composing readable arbitrary computations, include the construction of event handlers or monadic parsers.

LINQ Example

```
using System.Linq;
public void Linq1()  {
      int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

      var lowNums =
         from n in numbers
         where n < 5
         select n;

      foreach (var x in lowNums)
         System.Console.WriteLine(x);
 }
```

```
4
1
3
2
0
```

var can be assigned any type, here its an array of int. Below an *int.*

```
var i = 10;
```

LINQ Example

```
using System.Linq;
public void Linq1()  {
      int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

      var lowNums =
         from n in numbers
         where n < 5
         let y = n
         select y;

      foreach (var x in lowNums)
         System.Console.WriteLine(x);
  }
```

```
4
1
3
2
0
```

Exercise 1a

Give minor change to square each *int* in *numbers*

let defines variables local to LINQ statement.

LINQ Example/Exercise 1

```
using System.Linq;
class Program {
    static void Main() {
        string[] a = new string[] {"I", "J", "A", "K"};
        var sort = from s in a
                   orderby s ascending
                   select s;

        foreach (string c in sort)
            System.Console.WriteLine(c);
    }
}
```

A
I
J
K

Lambda Expressions

A lambda expression is an anonymous function used to create delegates or expression tree types. Use lambda expressions to write local functions that can be passed as arguments or returned as the value of function calls.
http://msdn.microsoft.com/en-us/library/bb397687.aspx

`x => x * x` is anonymous lambda expression.

`delegate int aFun(int i);` defines a signature.

`aFun myFun = x => x * x;` binds the lambda expression to `myFun`.

```
delegate int aFun(int i);

static void Main( ) {
    aFun myFun = x => x * x;
    int j = myFun(5);
}
```

j = 25

Lambda Expressions

A lambda expression is an anonymous function used to create delegates or expression tree types. Use lambda expressions to write local functions that can be passed as arguments or returned as the value of function calls.
http://msdn.microsoft.com/en-us/library/bb397687.aspx

`x => x * x` is anonymous lambda expression.

`delegate int aFun(int i);` defines a signature.

`aFun myFun = x => x * x;` binds the lambda expression to `myFun`.

```
delegate int aFun(int i);

static void Main( ) {
    aFun myFun = x => x * x;
    int j = myFun(5);
}
```

j = 25

Lambda Expressions LINQ

Lambda expression used in LINQ statements.

```
static void Main( ) {
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        var positive = numbers.Where<int>(n => n < 5);

        foreach (int c in positive) System.Console.WriteLine(c);
}
```

```
4
1
3
2
0
```

Combining operations

Exercise 1b: Output?

```
var square = numbers.Where<int>(n => n < 5).Select<int, int>(n => n * n);
```

Func

`Func<T, TResult>`

Encapsulates a method that has one parameter of type T and returns a value of the type specified by the TResult parameter.

```
Func<int, double> f ;
```

Defines *f* as a function of one *int* parameter, returning a *double* result.

```
f = delegate(int x) { return 3.0 * x; };
f = x => 3.0 * x;
```

Binds *f* to the anonymous function x => 3.0*x of one *int* parameter, returning a *double* result.

```
string example( int i, Func<int, string> f)
```

Defines a function *example* with first parameter an int and second parameter a function of one *int* parameter, returning a *string*.

```
Func<int[], double, string>
```

First parameter is *int[]*, second *double,* returns *string*.

Lambda Expressions

```
using System;
class Program   {
   static string example( int i, Func<int, string> f) {
      return f (i);
   }
   static void Main()  {
      Func<double, double> f = delegate(double x) { return 3 * x; };
      Console.WriteLine( f(4) );        // 12

      f = x => 3*x;
      Console.WriteLine( f(5) );        // 15

      Func<double, double> g = x => 3*x;
      Console.WriteLine(g(6));          // 18

      Console.WriteLine(example(5, x => string.Format("String = {0}", x)));
    }
}
```

Exercise 2 – Result of following?

Lambda Expressions – int filter

`element => element > 1` anonymous function

```
using System.Collections.Generic;
class Program {
   delegate bool predicate(int i);
   static void Main() {
      List<int> values = new List<int>() { 1, 3, 1, 2, 3 };
      List<int> result = filter( element => element > 1, values);
      foreach (int val in result) System.Console.WriteLine(val);
   }
   static List<int> filter(predicate p, List<int> L)  {
      List<int> result = new List<int>();
      foreach (int val in L)    if ( p(val) ) result.Add(val);
      return result;
   }
}
```

```
3
2
3
```

Exercise 3: Write the *map* functional for *List<int>*.

Lambda Expressions – Parameterized filter

```
using System.Collections.Generic;
class Program{
    delegate bool predicate<T>(T i);
    static void Main()    {
        List<int> values = new List<int>() { 1, 3, 1, 2, 3 };

        List<int> result = filter<int>(element => element > 1, values);

        foreach (int val in result) System.Console.WriteLine(val);
    }

    static List<T> filter<T>(predicate<T> p, List<T> L)    {
        List<T> result = new List<T>();
        foreach (T val in L)  if (p(val)) result.Add(val);
        return result;
    }
}
```

```
3
2
3
```

Lambda Expressions – Parameterized filter

```
using System.Collections.Generic;
class Program  {
    static void Main()  {
       List<int> values = new List<int>() { 1, 3, 1, 2, 3 };
       Functionals<int> fun = new Functionals<int>( );
       List<int> result = fun.filter(element => element > 1, values);
       foreach (int val in result) System.Console.WriteLine(val);
    }
}
public class Functionals<T> {
   public delegate bool predicate(T i);
   public List<T> filter(predicate p, List<T> L)    {
       List<T> result = new List<T>();
       foreach (T val in L)
          if (p(val)) result.Add(val);
       return result;
   }
}
```

```
3
2
3
```

Lambda Expressions – Parameterized map

`element => element * 2` anonymous unary function

```
using System.Collections.Generic;
class Program{
   delegate T unaryFunction<T>(T i);

   static void Main()    {
      List<int> values = new List<int>() { 1, 3, 1, 2, 3 };

      List<int> result = map<int>(element => element * 2, values);

      foreach (int val in result) System.Console.WriteLine(val);
   }

   static List<T> map<T>( unaryFunction<T> f, List<T> L)  {
      List<T> result = new List<T>();
      foreach (T val in L)    result.Add( f( val ) );
      return result;
   }
}
```

```
2
6
2
4
6
```

Exercise 4: Write the *map2* functional for *List<T>*.

Lambda Expressions – Parameterized map

`element => element * 2` anonymous unary function

```
using System.Collections.Generic;
class Program {
    static void Main() {
        List<int> values = new List<int>() { 1, 3, 1, 2, 3 };
        Functionals<int> fun = new Functionals<int>();
        List<int> result = fun.map(element => element * 2, values);
        foreach (int val in result) System.Console.WriteLine(val);
    }
}
public class Functionals<T> {
    public delegate T unaryFunction(T i);
    public List<T> map( unaryFunction f, List<T> L)  {
        List<T> result = new List<T>();
        foreach (T val in L)    result.Add( f( val ) );
        return result;
    }
}
```

```
2
6
2
4
6
```

Lambda Expressions – Generic map2

`(x, y) => x + y` anonymous *binary* function

```
    List<int> values1 = new List<int>() { 1, 3, 1, 2, 3 };
    List<int> values2 = new List<int>() { 1, 3, 1, 2, 3 };
    List<int> result = map2<int>( (x, y) => x + y, values1, values2);
    foreach (int val in result) System.Console.WriteLine(val);

delegate T binaryFunction<T>(T x, T y);

static List<T> map2<T>(binaryFunction<T> f, List<T> L1, List<T> L2)    {
    List<T> result = new List<T>();
    List<T>.Enumerator e = L2.GetEnumerator();
    foreach (T val1 in L1)       {
        e.MoveNext();
        T val2 = e.Current;
        result.Add(f(val1, val2));
    }
    return result;
}
}
```

2
6
2
4
6

Exercise 4 solution : Write the *map2* functional for *List<T>*.

Lambda Expressions – Generic map2

`(x, y) => x + y` anonymous *binary* function

```
List<int> values1 = new List<int>() { 1, 3, 1, 2, 3 };
List<int> values2 = new List<int>() { 1, 3, 1, 2, 3 };
Functionals<int> fun = new Functionals<int>();
List<int> result = fun.map2( (x, y) => x + y, values1, values2);

public class Functionals<T> {
  public delegate T binaryFunction(T x, T y);
  public List<T> map2(binaryFunction f, List<T> L1, List<T> L2)  {
      List<T> result = new List<T>();
      List<T>.Enumerator e = L2.GetEnumerator();
      foreach (T val1 in L1)     {
         e.MoveNext();
         T val2 = e.Current;
         result.Add( f(val1, val2) );
      }
      return result;
  }
```

2
6
2
4
6

Exercise 4 solution : Write the *map2* functional for *List<T>*.

Closures

A first-class function with free variables bound in the lexical environment.

```
class closures {
    static void Main()  {
        var inc = getInc();
        System.Console.WriteLine( inc( 5 ));
    }
    public static Func<int, int> getInc()   {       // increment assigned delegate
            int incValue = 1;                       // function
            Func<int, int> increment = delegate(int n)  {
                return n + incValue;
            };
            return increment;
    }
}
```

Func<T, TResult>
Func<int, int>

Exercise 5: Output?

incValue is bound in *increment* lexical scope.

Func<int, int> increment = delegate(int n) defines *increment* as function that has int parameter and returns int.

Closures binding functions

```
using System;
class closures {
   static void Main(string[] args)  {
      var mapInc = stagedMap(x => x + 1);          // Anon function: x => x + 1
      int[] L = { 1, 2, 3 };
      foreach (int val in mapInc(L)) Console.WriteLine(val);
   }

   public static Func<int[], int[]> stagedMap(Func<int, int> f) {
      Func<int[], int[]> map = delegate(int[] L)
      {
         for (int i = 0; i < L.Length; i++)
            L[i] = f( L[i] );
         return L;
      };
      return map;
   }
}
```

Func<T, TResult>
Func<int[], int[]>

Exercise 6: Output?

f is bound in *map* lexical scope.

Func<**int[], int[]**> map = delegate(int[] L) defines *map* as function that has int[] parameter and returns int[].

Closures using named delegates

```
class closures  {
   public delegate int unaryFunction(int i);
   public delegate int [] arrayFunction( int [] L );

   static void Main(string[] args)  {
       int[] L = { 1, 2, 3 };
      var mapInc = stagedMap(x => x + 1);
      System.Console.WriteLine(mapInc(L)[2]);
   }
   public static arrayFunction stagedMap( unaryFunction f )    {
      arrayFunction map = delegate(int[] L)    {
         for (int i = 0; i < L.Length; i++)
            L[i] = f( L[i] );
         return L;
      };
      return map;
   }
}
```

Exercise 7:
Output?

f is bound in *map* lexical scope.

Exercise 7.5 – Assume C# arrays grow dynamically (some other languages do) when a new index is used.

```
int [] L = {10, 20, 30};
L[3] = 40;
```

Give a similar

```
stagedFilter(x => x > 1)
```

Closure saves state

Closure saves computation state from one call to next, remembering old values and computing only new values.

```
fact 1
fact 2
fact 6
fact 24
Main 24
fact 120
fact 720
Main 720
```

```
class closures  {
    static void Main()   {
        var f = factFun();
        System.Console.WriteLine( "Main " + f(4) );
        System.Console.WriteLine( "Main " + f(6) );    // f(4) cont.
    }
    public static Func<int, int> factFun() {     // State returned
        int i = 1;
        int[] result = new int[100];
        result[0] = 1;
        for (int j = 1; j < 100; j++) result[j] = -1;
        Func<int, int> fact = delegate(int n) {
            while (i <= n && result[i] == -1) {
                result[i] = result[i - 1] * i;
                System.Console.WriteLine("fact " + result[i]);
                i++;
            }
            return result[n];
        };
        return fact;
    }
}
```

Exercise 7.6:
f(3) result?

Closures saving computation state

Closure saves computation state from one call to next, remembering old values and computing only new values of Fibonacci.

```
using System;
class closures {
   static int count = 0;

   static void Main()
   {
      var f = getFib();
      Console.WriteLine(f(20)+" "+count);
      count = 0;
      Console.WriteLine(f(18)+" "+count);
   }

   public static int fib(int n, int [] result)
   {
      count++;
      if (n == 0 || n == 1) return n;
      result[n-2] = fib(n - 2, result);
      result[n-1] = fib(n - 1, result);
      return result[n - 1] + result[n - 2];
    }
```

Output 6765 21890

2584 0 No new computation

```
  public static Func<int, int> getFib()   {
      int[] result = new int[100];
      result[0] = 0;
      result[1] = 1;
      for (int i = 2; i < 100; i++) result[i] = -1;

      Func<int, int> fibFun = delegate(int n)
      {
         if (result[n-2] == -1) result[n-2]=fib(n - 2, result);
         if (result[n-1] == -1) result[n-1]=fib(n - 1, result);
         return result[n-1]+result[n-2];
      };
      return fibFun;
   }
}
```

Recursive Lambda expressions

So far, all lambda expressions have been non-recursive, as below.

```
using System.Collections.Generic;
class Program{
   static void Main()    {
       List<int> values = new List<int>() { 1, 3, 1, 2, 3 };

       List<int> result = map<int>(element => element * 2, values);

       foreach (int val in result) System.Console.WriteLine(val);
    }

    static List<T> map<T>( Func<T, T> f, List<T> L)  {
       List<T> result = new List<T>();
       foreach (T val in L)    result.Add( f( val ) );
       return result;
    }
}
```

Recursive Lambda expressions

http://blogs.msdn.com/b/wesdyer/archive/2007/02/02/anonymous-recursion-in-c.aspx

But recursive lambda expressions are possible, as below.
Well, maybe not this way.

```
using System;
class One {
   static void Main() {
          Func<int, int> fib = n => n > 1 ? fib(n - 1) + fib(n - 2) : n;
    }
}

Error    1          Use of unassigned local variable 'fib'
```

Problem: trying to use *fib* on right side of = before it is definitely assigned.

Recursive Lambda expressions

Assign identifier *fib* first, can then reassign *fib* to the lambda expression.

But actually not recursive.

```
using System;
class Two {
    static void Main() {
        Func<int, int> fib = null;

        fib = n => n > 1 ? fib(n - 1) + fib(n - 2) : n;

        Console.WriteLine(fib(6));                    // Display 8
    }
}
```

Recursive Lambda expressions

Assign identifier *fib* first, then reassign *fib* to the lambda expression.

```
Func<int, int> fib = null;
fib = n => n > 1 ? fib(n - 1) + fib(n - 2) : n;

Func<int, int> fibCopy = fib;
Console.WriteLine(fib(6));                    // displays 8
Console.WriteLine(fibCopy(6));                // displays 8

fib = n => n * 2;
Console.WriteLine(fib(6));                    // displays 12
Console.WriteLine(fibCopy(6));                // displays 18
```

`fibCopy(6) = fib(5) + fib(4) = 5*2 + 4*2 = 18`

Side-effect - redefines *fib* referenced in the lambda expression of *fibCopy*.

Recursive Lambda expressions

Solution: Pass the recursive function to itself as a parameter.

(f, n) – f is the function

fib(fib, 6) – *fib* binds fib to *f* and 6 to *n*.

f(f, n - 1) becomes: fib(fib, 6 - 1)

```
class Three {
   delegate int Recursive(Recursive r, int a);

   static void Main()    {
            Recursive fib = (f, n) => n > 1 ? f(f, n - 1) + f(f, n - 2) : n;
            System.Console.WriteLine(fib(fib, 6));                    // Display 8
}
```

Can parameterize delegate.

Recursive Lambda expressions
Parameterized delegate

Pass the recursive function to itself as a parameter.

(f, n) – f is the function

fib(fib, 6) – *fib* binds fib to *f* and 6 to *n*.

f(f, n - 1) becomes: fib(fib, 6 - 1)

```
class Four {
   delegate R Recursive<A, R>(Recursive<A, R> r, A a);

   static void Main()    {
          Recursive<int, int> fib = (f, n) => n > 1 ? f(f, n - 1) + f(f, n - 2) : n;
          System.Console.WriteLine(fib(fib, 6));              // Display 8
}
```

Detour – Curried functions

curriedAdd takes an *int* and returns a function that takes an *int* and returns an *int*:

Func<int, Func<int, int>> curriedAdd = x => y => x + y;

var add5 = curriedAdd(5);

var assigns same type as

Func<int, int> add5 = curriedAdd(5);

```
class Five{
   static void Main() {
         Func<int, int, int> add = (x, y) => x + y;
         int a = add(2, 3);                    // a = 5

         Func<int, Func<int, int>> curriedAdd = x => y => x + y;
         int b = curriedAdd(2)(3);             // b = 5

         var add5 = curriedAdd(5);             // x bound to 5
         int c = add5(3);                      // c = 8
    }
}
```

Functional Summary

- LINQ
- Closures
- Lambda expressions
- Anonymous functions
- Curried functions