Exercises

1) Lets create a roman numerals converter.
   Read about roman numerals here: http://mathworld.wolfram.com/RomanNumerals.html
   I have created the following test class

```
public class TestRomanLiterals
{
  [TestCase(1, "I")]
  [TestCase(2, "II")]
  [TestCase(3, "III")]
  [TestCase(4, "IV")]
  [TestCase(5, "V")]
  [TestCase(6, "VI")]
  [TestCase(9, "IX")]
  [TestCase(27, "XXVII")]
  [TestCase(48, "XLVIII")]
  [TestCase(59, "LIX")]
  [TestCase(93, "XCIII")]
  [TestCase(141, "CXLI")]
  [TestCase(163, "CLXIII")]
  [TestCase(402, "CDII")]
  [TestCase(575, "DLXXV")]
  [TestCase(911, "CMXI")]
  [TestCase(1024, "MXXIV")]
  [TestCase(3000, "MMM")]
  public void TestRomenConversion(int number, string expected)
  {
     var arabic = new Arabic(number);

     Assert.AreEqual(expected, arabic.ToRoman());
  }
}
```

   a) So try start making the Arabic class, and implement the ToRoman() method
   b) Is it written in a functional style? If not try to implement this in a way without any mutable state.

2) Lets implement an immutable data structure. This is a very common pattern i FP. Having immutable data structures, means that we can use the instance in a parallel environment, since we never change any value, but always copy.
Immutable data structures is a whole branch of FP. We are going to implement a list.
Create a project with a the following three classes

```
public abstract class List<TA> {}
public sealed class Nil<TA> : List<TA> {}
public sealed class Cons<TA>: List<TA>
{
  public readonly TA X;
  public readonly List<TA> Tail;
}
```

a) Before implementing anything make sure you understand this data structure.

All mutating methods we are implementing will be static methods in the List class.

b) Implement a Init method[1]

```
public static List<TA> Init(params TA[] a)
```

c) Adding elements to the front of the list is cheap, implement

```
public static List<TA> SetHead(List<TA> l, TA element)
```

Adding elements to the back of a single chained list, is expensive

d) Try implementing these two functions

```
public static List<TA> Drop(List<TA> l, int n)
public static List<TA> DropWhile(List<TA> l, Func<TA, bool> f)
```

3) With the methods you now have on your List the following methods can quite easily be implemented:
   a) Count number of elements
   b) Given a list of int [1, 2, 3, 4, 5, 6, 7, 8] calculate list containing their string representation [“1”, “2”, “3”, … “8”]
   c) Generate a list of even numbers
   d) Calculate the sum of the elements in the list.

[1] Hint: Think recursive

4[2]) Given a string like this

`"7 3 - 2 1 + *"`

You can use the GoF Interpreter pattern (https://en.wikipedia.org/wiki/Interpreter_pattern) to quite easily translate the expression to a sum.

The same can be done with lambdas by “translating” the expressions objects to functions. Try implementing the interpreter pattern both with objects and lambda.

The below skeleton code can be used basis for GoF pattern and the Evaluate and IsOperator can also be used to implement a more function version.

```
public class Interpreter
{
  public interface Expression
  {
     int Intepret();
  }

  public int Evaluate(string expression)
  {
     Stack<Expression> expressions = new Stack<Expression>();
     foreach (var str in expression.Split(" "))
     {
        if (IsOperator(str))
        {
           throw new NotImplementedException();
        }
        else
        {
           throw new NotImplementedException();
        }
     }
     throw new NotImplementedException();
  }

  private bool IsOperator(string s)
  {
     if (s.Equals("+") || s.Equals("-") || s.Equals("*"))
     {
        return true;
     }

     return false;
  }
}
```

[2] A bit hard