C#/.Net project cluster **ITU, May 2006**

Exercises
for Wednesday 3 May 2006

2005-04-26

There are probably too many exercises here. When you get tired of enumerables, jump to the last exercise so you get to use nullable types also.

Exercise C# 1 The purpose of this exercise is to illustrate the use of delegates and especially anonymous method expressions of the form `delegate(...) { ... }`.

Get the file `http://www.itu.dk/people/sestoft/csharp/IntList.cs`. The file declares some delegate types:

```
public delegate bool IntPredicate(int x);
public delegate void IntAction(int x);
```

The file further declares a class IntList that is a subclass of .Net's List<`int`> class (which is an arraylist; see C# Precisely section 24.4). Class IntList uses the delegate types in two methods that take a delegate as argument:

- `list.Act(f)` applies delegate `f` to all elements of `list`.
- `list.Filter(p)` creates a new IntList containing those elements `x` from `list` for which `p(x)` is true.

Add code to the file's `Main` method that creates an IntList and calls the `Act` and `Filter` methods on that list and various anonymous delegate expressions. For instance, if `xs` is an IntList, you can print all its elements like this:

```
xs.Act(Console.WriteLine);
```

This works because there is an overload of `Console.WriteLine` that takes an `int` argument and therefore conforms to the `IntAction` delegate type.

You can use `Filter` and `Act` to print only the even list elements (those divisible by 2) like this:

```
xs.Filter(delegate(int x) { return x%2==0; }).Act(Console.WriteLine);
```

Explain what goes on above: How many IntList are there in total, including `xs`?

Further, use anonymous methods to write an expression that prints only those list elements that are greater than or equal to 25.

An anonymous method may refer to local variables in the enclosing method. Use this fact and the `Act` method to compute the sum of an IntList's elements (without writing any loops yourself).

Note: If you have an urge to make this exercise more complicated and exciting, you could declare a generic subclass MyList<T> of List<T> instead of IntList, and make everything work for generic lists instead of just IntLists. You need generic delegate types Predicate<T> and Action<T>, but in fact these are already declared in the .Net System namespace.

Exercise C# 2 This exercise and the next one explore some practical uses of enumerables and the `yield` statement.

Declare a static method `ReadFile` to read a file and return its lines as a sequence of strings:

```
static IEnumerable<String> ReadFile(String fileName) { ... }
```

C# Precisely section 22.4 describes TextReader and example 153 shows how to create a StreamReader by opening a file. (The good student will of course use a `using` statement — C# Precisely section 13.10 — to bind the TextReader to make sure the file gets closed again, even in case of errors).

The `ReadFile` method should read lines from the TextReader, using the `yield return` statement to hand the lines to the 'consumer' as they are produced. The consumer may be a `foreach` statement such as
`foreach (String line in ReadFile("foo.txt")) Console.WriteLine(s);`.

Exercise C# 3 Declare a static method `SplitLines` that takes as argument a stream of lines (strings) and returns a stream of the words on those lines (also strings)

```
static IEnumerable<String> SplitLine(IEnumerable<String> lines) { ... }
```

C# Precisely example 191 shows how a regular expression (of class System.Text.RegularExpressions.Regex) can be used to split a string into words, where a 'word' is a non-empty contiguous sequence of the letters a–z or A–Z or the digits 0–9.

The `SplitLine` method should use a `foreach` loop to get lines of text from the given enumerable `lines`, and use the `yield return` statement to produce words.

It should be possible to e.g. find the average length of words in a file by combining the two methods:

```
int count = 0, totalLength = 0;
foreach (String word in SplitLines(ReadFile "foo.txt")) {
  count++;
  totalLength += word.Length;
}
double averageLength = ((double)totalLength)/count;
```

Note that in this computation, only a single line of the file needs to be kept in memory at any one time. In particular, the call to `ReadFile` does not read all lines from the file before `SplitLines` begin to produce words. That would have been the case if the methods had returned lists instead of enumerables.

Exercise C# 4 The purpose of this exercise and the next one is to emphasize the power of enumerables and the `yield` and `foreach` statements.

Declare a generic static method `Flatten` that takes as argument an array of IEnumerable<T> and returns an IEnumerable<T>. Use `foreach` statements and the `yield return` statement. The method should have this header:

```
public static IEnumerable<T> Flatten<T>(IEnumerable<T>[] ebles) { ... }
```

If you call the method as shown below, you should get 2 3 5 7 2 3 5 7 2 3 5 7:

```
IEnumerable<int>[] ebles = new IEnumerable<int>[3];
ebles[0] = ebles[1] = ebles[2] = new int[] { 2, 3, 5, 7 };
foreach (int i in Flatten<int>(ebles))
  Console.Write(i + " ");
```

Exercise C# 5 (If you enjoy challenges) Redo the preceding exercise without using the `yield` statement.

Exercise C# 6 The purpose of this exercise is to illustrate computations with nullable types over simple types such as `double`.

To do this, implement methods that work like SQL's aggregate functions. We don't have a database query at hand, so instead let each method take as argument an IEnumerable<`double?`>, that is, as sequence of nullable `double`s:

- `Count` should return an `int` which is the number of non-`null` values in the enumerable.
- `Min` should return a `double?` which is the minimum of the non-`null` values, and which is `null` if there are no non-`null` values in the enumerable.
- `Max` is similar to `Min` and there is no point in implementing it.
- `Avg` should return a `double?` which is the average of the non-`null` values, and which is `null` if there are no non-`null` values in the enumerable.
- `Sum` should return a `double?` which is the sum of the non-`null` values, and which is `null` if there are no non-`null` values. (Actually, this is weird: Mathematically the sum of no elements is 0.0, but the SQL designers decided otherwise. This design mistake will also make your implementation of `Sum` twice as complicated as necessary: 8 lines instead of 4).

When/if you test your method definitions, note that `null` values of any type are converted to the empty string when using `String.Format` or `Console.WriteLine`.