UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFR08014 INFORMATICS 1 - OBJECT-ORIENTED PROGRAMMING

Wednesday 1st May 2013

14:30 to 17:30

INSTRUCTIONS TO CANDIDATES

1. **Note that all questions are compulsory.**
2. **This is an Open Book exam.**

Convener: J Bradfield
External Examiner: A Preece

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

Question 1

In this question you will implement part of a system to support the planning of composite documents, e.g. books made up of chapters.
Important: your work will be marked partly by automated tests, so you must follow the instructions precisely. For example, where your program produces strings, use exactly the format specified.

(a) Implement a class `Document` as follows.

`Document` must have private attributes `name : String` and `pages : int` and the following public interface:

`public class Document`

`Document(String name, int pages)`	*constructor*
`String getName()`	*getter*
`void setName(String name)`	*setter*
`int getPages()`	*getter*
`void setPages(int pages)`	*setter*
`String toString()`	*return "Name (Pages)" e.g. "Stamp Collecting (15)"*

[15 marks]

(b) Implement the class `Book` to extend `Document`. This class should have an additional private attribute `chapters` of type `ArrayList<Document>`, initialised, in the constructor, to be empty. (Remember you will need the line

```
import java.util.ArrayList;
```

at the top of your file.) Provide a getter and a setter method for this new attribute. Override the method `toString()` so that it gives the book's own name, then on a new line by itself gives "Contains:", then lists the documents that are in the `Book`'s chapters attribute exactly as the document's own `toString()` does, each on a new line. For example, if `"`All about Stamps", having 300 pages, includes chapter "Stamp Collecting" which has 15 pages, then sending `toString()` to an object representing "All about Stamps" should result in:

```
All about Stamps (300)
Contains:
Stamp Collecting (15)
```

Do not concern yourself about whether the page numbers reported are sensible – e.g., assume it is fine for a Book to contain Chapters totalling more pages than the Book has. It is possible that a Book might contain another Book recursively – that is, that one of the Documents in the chapters attribute might itself be a Book with its own chapters – but your code should not take any specific action either to prevent this or to handle such a case specially.

In summary `Book` has this public interface, with behaviour as explained above:

```
public class Book
-------------------------------------------------------------------
                      Book(String name, int pages)
  ArrayList<Document> getChapters()
                 void setChapters(ArrayList<Document> chapters)
               String toString()
-------------------------------------------------------------------
```

[20 marks]

(c) Next, implement class `Position`. This has two private integer attributes, `pageNo` and `lineNo`, intended to represent a line in a document (though you will not be implementing any connection between this class and the ones you wrote earlier in this question). E.g. if a `Position` has `pageNo==3` and `lineNo==4`, it represents line 4 of page 3 of the document. The value of `lineNo` will be between 1 and 25 inclusive. In contrast, `pageNo` may be any integer, including negative; documents occasionally use negative page numbers for their preambles.

Your class must have a zero-argument constructor that sets `pageNo` to 1 and `lineNo` to 1, and getters and setters for its attributes. If the setter for `lineNo` is given an argument outside the acceptable range, it should simply do nothing: that is, it must not change the object, but it must not report this in any way as a mistake.

Your class must also have a method `advance` taking a positive integer `n` and returning nothing. It modifies the state of this Position object by moving the position it represents forward by `n` lines, going on to the next page when necessary. Again, if the argument is negative or zero, the method should simply do nothing.

Finally the `toString` method should return "`Page:  3 Line:  4`" if `pageNo` is 3 and `lineNo` is 4, and so on.

In summary `Position` has this public interface, with behaviour as explained above:

```
public class Position
-------------------
        Position()
    int getPageNo()
   void setPageNo(int n)
    int getLineNo()
   void setLineNo(int n)
   void advance(int n)
 String toString()
-------------------
```

[15 marks]

The files that you must submit for this question are the following:

- `Document.java`
- `Book.java`
- `Position.java`

Question 2

Note: in this question code that fails to compile will receive no credit.

In each of parts (a)–(c) below, you will be asked to supply the body to a method inside a class `Lens`. You will be given a skeleton file for this class. You should add your definitions of the methods at the points marked as follows:

```
// ADD CODE HERE
```

In software engineering, a lens manages the consistency between a *source* piece of data, $s \in S$, and an abstracted *view* of it, $v \in V$. It comprises three functions:

- get : $S \to V$ which, given a source, returns the view of that source (this typically involves throwing some information away);
- create : $V \to S$ which, given a view, returns a source of which this could be the view (this typically involves filling in some missing information with default values)
- put : $S \times V \to S$. Given a source and an *updated* version of the view of that source, this returns an updated version of the source.

In this question you will implement and test the three functions `get`, `create` and `put` for a lens between arrays of `String`s and arrays of `Pair`s, where each `Pair` comprises a character (type `char`) and a natural number (type `int`). You are given the class `Pair`: study it.

Note: it is expected that you may not be familiar with the Java base type `char`. You should consult the Java documentation whenever you need to. Hint: you are likely to need the `isUpperCase` static method from `Character`, and the `charAt` method of `String`.

(a) In the class `Lens`, implement the public static method

```
Pair[] get(String[] source)
```

that returns an array of Pairs the same length as the input array of Strings. Each Pair corresponds to the String at the same position. The Pair contains the first character of the string and the number of upper case letters in the string. You may assume that each String has length greater than 0.

Expected behaviour:

```
get(new String[] {"foo", "BAR", "frObOz"})
should return the array of Pairs
{['f',0],['B',3],['f',2]}
```

[*10 marks*]

(b) In the class `Lens`, implement the public static method

```
String[] create(Pair[] view)
```

that returns an array of `String` the same length as the input array of `Pair`s. Each `String` corresponds to the `Pair` at the same position. The `String` must consist of the initial character from the `Pair`, followed by a number of 'X's given by the integer from the `Pair`. Add this number of 'X's even if the first character is upper case: see the first `Pair` in the example below.

Expected behaviour:

```
create (new Pair[]
        {new Pair('F',2), new Pair('b',0), new Pair('f',5)})
should return the array of Strings
{"FXX", "b", "fXXXXX"}
```

[*10 marks*]

(c) In the class `Lens`, implement the public static method

```
String[] put(String[] oldSource, Pair[] newView)
```

that expects `oldSource` and `newView` to have the same length (say n) – this check is done for you in the code provided – and returns an array of strings of length n, which is the new source.

Each new source element must have *at least* the number of upper case letters indicated in the corresponding element of the new view. If the corresponding element of the old source already has at least that number of upper case letters, then do not alter it: the new source must contain exactly the same element as the old source. If the corresponding element of the old source has *fewer* upper case letters than indicated in the new view element, the new source element is formed by appending just enough 'X's to the old source element.

Expected behaviour:

```
put (new String[] {"foo", "BAR", "frObOz"},
     new Pair[] {new Pair('f',2), new Pair('b',0), new Pair('f',5)})
should return the array of Strings
{"fooXX", "BAR", "frObOzXXX"}

put(new String[] {"foo", "bar", "FROBOZ"},
    new Pair[] {new Pair('f',1), new Pair('b',6), new Pair('f',3)})
should return the array of Strings
{"fooX", "barXXXXXX", "FROBOZ"}
```

If the first character of a `Pair` in `newView` is not the first character of the corresponding string in `oldSource`, your code should behave the same way as if the length check failed. Use the provided length-checking code as an example, modifying the error message to "First characters don't match". [*20 marks*]

(d) One important law that some lenses obey is called PutGet. In this part you will implement a class `InvestigateLens` that tests this law on your `Lens` class.

(i) The PutGet law says: for every source s, `put(s,get(s)) = s`.

Implement a static method of `InvestigateLens` called `checkPutGet` that takes a source (type String[]) and returns true if and only if the PutGet law holds for this particular value. [*5 marks*]

(ii) Implement a `main` method for `InvestigateLens` that invokes the method you implemented in parts (i) on the following inputs:

source s
{"FOO"}
{"foo", "BAR", "frObOz" }
{"foo", "bar", "FROBOZ" }

Your method must print `true` or `false` to standard output to show whether your lens passes each test, and finally, whether it passes all three tests and therefore may (as far as these tests show) obey the law. Each result must be on a new line, and nothing else must be printed. For example, if your lens fails PutGet on all three given inputs and therefore definitely does not satisfy the PutGet law for all inputs, your output should be

```
false
false
false
false
```

[*5 marks*]

The files that you must submit for this question are the following:

- `Lens.java`
- `InvestigateLens.java`

Final Checklist

Here is a complete list of all the files that you must submit for this exam:

```
Document.java
Book.java
Position.java
Lens.java
InvestigateLens.java
```