UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFR08014 INFORMATICS 1 - OBJECT-ORIENTED PROGRAMMING

Tuesday 5 th May 2015

09:30 to 12:30

INSTRUCTIONS TO CANDIDATES

1. **Note that all questions are compulsory.**
2. **Remember that a file that does not compile, or does not pass the simple JUnit tests provided, will get no marks.**
3. **This is an Open Book exam. You may bring in your own material on paper, and/or on a USB stick. No other electronic devices are permitted.**
4. **CALCULATORS MAY NOT BE USED.**

Convener: D. K. Arvind
External Examiner: C. Johnson

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

1. You are given, in file `Seller.java`, the code of a class `Seller` with the following public interface:

public class Seller		
	`Seller(String n)`	constructor
`double`	`getWeightSold()`	
`void`	`sale(double d)`	
`String`	`toString()`	

Examine the code of this class. You will see that it keeps a running total of the total weight sold by this seller, adding to it each time the `sale` method is used.

Your task is to implement the subclass `BonusSeller`, representing a special kind of `Seller`.

(a) Define the class `BonusSeller`, extending `Seller`. It should have a private instance variable `sales` of type `HashMap<String, Double>`, representing the names of customers and the *total* weight sold to each customer (over all sales to that customer). [*10 marks*]

(b) Write a public constructor for `BonusSeller`. Your constructor must, like the constructor of `Seller`, take a `String n`, and must pass this to `Seller`'s constructor. You must initialise `sales` with an initial capacity for 10 entries. [*10 marks*]

(c) Write a public instance method `sale`, taking a `String` representing the name of the customer and a `double` (not `Double`) representing how much the customer has bought on this occasion. Invoke the superclass's `sale` method, and then add this sale to the appropriate entry in `sales`. Make sure your code works correctly whether or not a previous sale has been made to this customer. You are not required to do any checks on the string and double given as arguments. [*10 marks*]

QUESTION CONTINUES ON NEXT PAGE

(d) Write a public instance method `toString`, taking no argument and returning a `String` representation of this object, exactly as follows. The string must begin with the string that is returned by `Seller`'s version of `toString` (even if the developers of that method decide to change it later, i.e., don't just duplicate the current behaviour). Then on a new line by itself, the string must have "Sales per customer:". Then list each customer on a line by itself. The order in which customers are listed does not matter. For each one, give the name, a colon, the total amount sold to that customer *rounded to the nearest integer* (assume no customer buys enough to overflow the `int` type!), then "kg". Here is an example of a possible result string:

```
Charles
Sales per customer:
Monsanto:13kg
ICI:35kg
```

[*10 marks*]

(e) Write a public instance method `calculateBonus`, taking no argument and returning an `int` representing the percentage bonus the seller will receive, calculated as follows. If there are fewer than 5 customers to whom the seller has sold at least 20kg, then return 0. If there are at least 5, but fewer than 10, then return 5. If there are at least 10, return 7. [*10 marks*]

In summary `BonusSeller` has this public interface (omitting what is inherited from `Seller`), with behaviour as explained above:

```
 public class BonusSeller
-----------------------------------
        BonusSeller(String n)
   void sale(String c, double d)
 String toString()
    int calculateBonus()
-----------------------------------
```

The file you must submit for this question is `BonusSeller.java`. Before you submit, check that it compiles and passes the basic JUnit tests provided, otherwise it will get 0.

2. **Hint:** although this question can be done "from first principles", there are methods in the standard library classes `System`, `Arrays` and `Collections` that may save you a considerable amount of work. Full marks are available both for solutions that use them, and for solutions that do not.

   This question relates to the Rabbit sequence, also known as the Golden String. It is a curious mathematical object, related to the more famous Fibonacci sequences. However, you are not expected to have any prior knowledge of any of these. The Rabbit sequence is built up via a sequence of sequences, where each sequence in turn gives a longer prefix of the (infinite) Rabbit sequence. The first few sequences are:

   ```
   [1]
   [1, 0]
   [1, 0, 1]
   [1, 0, 1, 1, 0]
   [1, 0, 1, 1, 0, 1, 0, 1]
   ```

   Each sequence, after the first two, is constructed by taking the previous sequence, and appending to it the sequence before that. For example, [1, 0, 1] is constructed by starting with the previous sequence, [1, 0], and appending the sequence before that, [1].

   (a) Implement a class `Rabbit` as follows. Note that the type `Integer` is (oddly, you may think, but take this as a customer requirement) used for the data in the Rabbit sequences. Indexes into the arrays are, of course, of type `int`, as usual.

   Your class should have two private instance variables:

   - `r`, an array of arrays of `Integer`s, which will represent the sequence of sequences as far as it has been calculated within this object;
   - `n`, an `int`, which will represent the number of sequences to be stored in this object. This will be at least 2; you are only required to check for this where specifically indicated below. *[5 marks]*

   (b) Write a public constructor, taking an `int n`, which stores `n` in the appropriate instance variable and initialises `r`, allocating space for `n` arrays. Note that the constructor should not allocate any space within any of these `n` arrays; that will be done in the `init` method. *[5 marks]*

   (c) Write a public instance method `init`, which takes no arguments, and populates the instance variable `r` with the first `n` Rabbit sequences, as described above. It should return no result. If `n` is less than 2, `init` should do nothing. *[10 marks]*

(d) Write a public instance method `toString`, which returns a `String` representing the sequence of sequences currently held in `r`, in the format given above (each sequence on a new line, in square brackets, with entries separated by a comma and one space, as shown). [*10 marks*]

(e) Write a public instance method `subsequenceIndex`, which takes an array of `Integers` (the "target") and returns an `int`. The value returned should be -1 if the target is not found in the longest Rabbit sequence contained in this object. Otherwise, it should contain the smallest index, within that longest sequence, at which the target appears. For example, if `subsequenceIndex([1,1,0])` is sent to a `Rabbit` object containing the sequences listed above, the result should be 2. The result of `subsequenceIndex([1,1,1])` should be -1. [*10 marks*]

(f) Finally, write a public static method `main` with the usual header. It should:

- expect, as command line argument, an integer that is at least 2;
- use that integer to construct a `Rabbit` object;
- initialise that object using the `init` method;
- print the object to standard output;
- check for the occurrence of subsequences [1,1,0] and [1,1,1]. Report the output simply by printing each integer returned by the `subsequenceIndex` method to standard output on its own line: do not print any explanatory text.

If the command line input is an integer that is less than 2, the program should do nothing. You are not required to handle the case where no integer is input.

Thus, for example, if the program is run from the command line as

```
java Rabbit 3
```

the output should be

```
[1]
[1, 0]
[1, 0, 1]
-1
-1
```

[*10 marks*]

The file you must submit for this question is `Rabbit.java`. Before you submit, check that it compiles and passes the basic JUnit tests provided, otherwise it will get 0.