Inf1 Object-Oriented Programming 2011/12

Mock Programming Exam

1. Note that all questions are compulsory.
2. Different questions may have different numbers of total marks. Take note of this in allocating time to questions.
3. This is an Open Book exam.

Question 1

In each of parts (a)–(c) below, you will be asked to supply the body to a method inside a class. There will be a separate class for each part, named `OneA`, `OneB` and `OneC` respectively. You will be given a skeleton file for each of these classes, and the skeleton will contain the appropriate method declaration. You should add your definitions of the methods at the points marked as follows:

```
    // ADD CODE HERE
```

(a) The *geometric mean* G is similar to the arithmetic mean, except that the numbers are multiplied and then the n^{th} root (where n is the count of numbers in the set) of the resulting product is taken. More formally, given positive real numbers $x_1, x_2, ..., x_n$, the geometric mean is defined to be

$$G = \sqrt[n]{x_1 x_2 \cdots x_n} = \prod_{i=1}^{n} (x_i)^{\frac{1}{n}}.$$

For example, the geometric mean of 2 and 8 is the *square root* of their product ($\sqrt{2 \times 8} = 4$), while the geometric mean of 1, 2, and 3 is the *cube root* of their product ($\sqrt[3]{1 \times 2 \times 3} = 1.817$ (to 3 dp).

In the class `OneA`, implement the static method

```
double geometricMean(int[] nums)
```

to compute this function.

Hint: To calculate the n^{th} root of a number k, use the fact that $\sqrt[n]{k} = k^{\frac{1}{n}}$; that is, evaluate the Java expression `Math.pow(k, 1.0/n)`.

Expected behaviour:

```
geometricMean(new int[] {1, 2, 3}) -> 1.8171205928321397
geometricMean(new int[] {2, 8}) -> 4.0
geometricMean(new int[] {3, 5, 7, 9}) -> 5.544443371329424
```

Note that the function you define may give slightly different output than these examples after the first couple of decimal places. *[10 marks]*

(b) In the class `OneB`, implement the following static method:

```
int longestSeq(int[] nums, int n)
```

Given an input array `nums` of integers, this method returns the length of the longest continuous sequence of `n`s in `nums`. If the input sequence is empty or if it contains no occurrences of `n`, the method should return 0. For simplicity, we will assume that `nums` only contains 0s or 1s.

Expected behaviour:

```
longestSeq(new int[] {1, 1, 0, 0, 1, 0, 1, 1, 1}, 0) ->  2
longestSeq(new int[] {1, 1, 0, 0, 1, 0, 1, 1, 1}, 1) ->  3
longestSeq(new int[] {1, 1, 1, 1}, 0) -> 0
longestSeq(new int[] {}, 1) -> 0
```

[15 marks]

(c) Given a string `s` of alphabetic characters, the overall task is to detect all substrings of length 3 in `s` and also to find out how many times each such substring occurs in `s`.

The task is broken down into subparts, each of which involves implementing the body of a static method within the class `OneC`.

(i) Implement the static method

```
ArrayList<String> findSubstrings(String s, int len)
```

which given an input string `s` and an integer `len`, finds all substrings of `s` of length `len`. You can assume that the argument `len` always has values ≥ 1. The output of `findSubstrings()` may well contain duplicates if the input contains the same characters in multiple positions. If `len` is greater than the length of `s`, then the method should return an empty `ArrayList<String>`.

Hint: Use the string method `s.substring(int beginIndex, int endIndex)` to find the required substrings of string `s`. If `endIndex - beginIndex` is greater than `s.length()`, then the `substring` method will throw an error, so you should make sure that you only let its arguments take admissible values.

Expected behaviour:

```
findSubstrings("abcc", 2) -> [ab, bc, cc]
findSubstrings("abcd", 4) -> [abcd]
findSubstrings("abc", 4) -> []
findSubstrings("aa", 1) -> [a, a]
```

[9 marks]

(ii) Implement the following static method:

```
void increment(HashMap<String, Integer> map, String s)
```

Assume that the `HashMap` argument to `increment()` is used to store a *frequency distribution*, where the keys of the `HashMap` are strings and the values are counts of occurrences. Calling `increment(map, s)` should increase the count for string `s` in `map`. More specifically, if `s` is not already a key in `map`, then `increment()` should put it into `map` with value 1; if `s` *is* already a key in `map`, then `increment()` should update the current value of `s` by 1. One simple way of checking whether a key `k` occurs in a `HashMap map` is to call `map.containsKey(k)`.

It is important to remember that `increment()` has no return value; instead it modifies the mappings in `map` directly.

Expected behaviour:

```
// current value of freq:
// HashMap<String, Integer> freq == {a=1}
increment(freq, "a");
increment(freq, "b");
// new value of freq:
// HashMap<String, Integer> freq == {a=2, b=1}
```

Note that we use {`a=1`} to represent a `HashMap` which assigns the value `1` to key `"a"`.

[*6 marks*]

(iii) Finally, implement the static method

```
HashMap<String, Integer> findStringCounts(String s)
```

by combining your two previous methods.

In order to implement `findStringCounts`, you should first create a local variable `freq` of type `HashMap<String, Integer>` to store the required frequency distribution. Next call your method `findSubstrings(s, 3)` to get an `ArrayList` of all substrings of `s` of length 3. Then call `increment(freq, sub)` to update `freq` with the counts of all substrings `sub` that you have found. Finally, return the value of `freq`.

Expected behaviour:

```
findStringCounts("abcdabcd")) -> {abc=2, dab=1, bcd=2, cda=1}
findStringCounts("abc")) -> {abc=1}
findStringCounts("ab")) -> {}
findStringCounts("XXXX")) -> {XXX=2}
```

Remember that the order in which a `HashMap` stores its key-value pairs is not important. If you inspect the value of the `HashMap` returned by your method, the order in which it stores its key-value pairs may differ from the examples shown above.

[*4 marks*]

(d) Create a class `QuestionOneTester` with a single `main()` method. Inside `main()`, add calls to the static methods

```
OneA.geometricMean()
OneB.longestSeq()
OneC.findStringCounts()
```

that you implemented for parts (a)–(c) above, in order to test that your implementations produce the correct results. (Remember that for a client program to call a static method from an external class, the method name must be qualified by the class name, as shown.) You are recommended to have your tests simply print out the value of the methods for some appropriate input arguments. Write one such test for each of the three methods specified. [*6 marks*]

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

(a) `OneA.java`

(b) `OneB.java`

(c) `OneC.java`

(d) `QuestionOneTester.java`

Question 2

This question involves building a simple model of chemical elements and molecules. The input data for the task is a short list of chemical elements, shown below in comma-separated value (CSV) format, where the column headings are element name, atomic number, symbol, and weight:

```
Hydrogen,1,H,1.01
Carbon,6,C,12.01
Nitrogen,7,N,14.01
Oxygen,8,O,16
Phosphorus,15,P,30.98
Sulfur,16,S,32.06
Potassium,19,K,39.1
```

This data is made available to you in the file `elements.csv`. Later in this question, you will need to process strings of this file, and it is useful to note that there are no spaces surrounding the comma.

First, you will construct a simple data type of chemical `Element`. Next, you will build a `Table` data type for holding a list of `Element` objects. Finally, you will build a chemical `Molecule` data type that represents the way in which elements can combine. Each of these sub-tasks is discussed in greater detail below.

A file `ChemistryKit.java` is provided to help you test your code and you may find it useful to look at this before starting your implementation.

(a) Here is the API for the `Element` data type:

`public class Element`		

	`Element(String element,`
	`int atomicNum,`
	`String sym,`
	`double weight)`
		constructor

`String`	`getElement()`	*the element's common name*
`int`	`getAtomicNum()`	*the element's atomic number*
`String`	`getSym()`	*the element's symbol*
`double`	`getWeight()`	*the element's weight*
`String`	`toString()`	*a string representation of the element*

Implement the class `Element` so that it meets the specified API.

An element will be initialized like this:

```
Element c = new Element("Carbon", 6, "C", 12.01);
```

In this case, the return value of `c.toString()` should look as follows:

```
Element(Carbon, 6, C, 12.01)
```

[*10 marks*]

(b) The next task is to implement the `Table` data type using this API:

`public class Table`	
`Table(String fn)`	*constructor*
`Table()`	*constructor*
`void readFile(String fn)`	*parse a* CSV *file into a list of elements*
`Element lookup(String sym)`	*given the symbol of an element, return the element*
`void display()`	*print out the elements in the table*

Implement the class `Table` so that it meets the specified API.

(i) As part of your class, create an `ArrayList<Element> table` to hold a list of `Element` objects.

The constructor `Table(String fn)` should call the instance method `readFile(String fn)` (defined below) to read the contents of a file. You are not expected to check that the input file is well-formed. The constructor `Table()` should invoke `Table(String fn)` to read the contents of the file `elements.csv`. [*5 marks*]

(ii) Implement the static method

```
readFile(String fn)
```

which takes a filename as argument and reads the file line-by-line. You should use the command `In file = new In(fn);` to read in the file. In order to process the file line-by-line, use a loop of the form

```
while (!file.isEmpty()){...}.
```

Then inside the loop, read each line of the file with the command `String line = file.readLine()` (of course, you can use any variable name you like instead of `line`). Convert each line into an array of type `String[]` so that you can identify and extract the information required to initialize a new `Element` object, and add that object to `table`.

Hint: You can use the string method `split(",")` to split a line on the comma character. [*5 marks*]

(iii) The instance method `lookup(String sym)` should take a string like `"C"` and return the `Element` object which has this string as its symbol. If there is no such element, the method should return `null`. [*5 marks*]

(iv) The instance method `display()` should iterate through the the elements in `table` and print them in sequence to the terminal. The output should look as follows:

```
Element(Hydrogen, 1, H, 1.01)
Element(Carbon, 6, C, 12.01)
Element(Nitrogen, 7, N, 14.01)
Element(Oxygen, 8, O, 16.0)
Element(Phosphorus, 15, P, 30.98)
Element(Sulfur, 16, S, 32.06)
Element(Potassium, 19, K, 39.1)
```

[*5 marks*]

(c) Consider a molecule such as water, with chemical formula H_2O. In our simple model of molecules, it suffices to capture the fact that water contains the elements hydrogen and oxygen in the ratio 2:1. We will represent this internally as a `HashMap<Element, Integer>` which maps an atom to the number of times it occurs in the molecule. That is, the `Molecule` water will hold the data {`h=2, o=1`}, where `h` and `o` are the `Element`s corresponding to hydrogen and oxygen respectively.

Here is the API for the `Molecule` data type:

`public class Molecule`		
	`Molecule()`	*constructor*
`void`	`addAtom(String sym, int num)`	*add `num` atoms of element with symbol `sym` to the molecule*
`void`	`addAtom(String sym)`	*add one atom of element with symbol `sym` to the molecule*
`ArrayList<Element>`	`atoms()`	*list of elements in the molecule*
`double`	`weight()`	*atomic weight of the molecule*

Implement the class `Molecule` so that it meets the specified API. You do *not* need to define the constructor for your class, since the default no-argument constructor added by Java will suffice.

(i) Define the following instance variables for the class `Molecule`: (i) a variable `structure` of type `HashMap<Element, Integer>` to hold information about the component atoms; (ii) a variable `table` of type `Table` which is assigned an object created by calling the `Table()` no-argument constructor. Recall that a `Table` object initialized in this way will read the contents of the file `elements.csv`. Consequently, instance methods in a `Molecule` will be able to access information about `Element`s by calling public methods of `table`.

In addition, implement the instance methods `addAtom(String sym, int num)` and `addAtom(String sym)`. To illustrate the first of these, given a `Molecule m`, the method call `m.addAtom("C", 3)` should carry out a look-up to determine that `"C"` is the symbol of the carbon `Element`, and will update `structure` to reflect that the molecule constains three atoms of carbon. You can assume that these only get called once for each distinct element that is added to a molecule. As indicated in the API above, if `sym` is of type `String`, then `addAtom(sym)` should return the same value as `addAtom(sym, 1)`. [*8 marks*]

(ii) Implement the instance method `atoms()`. This should return all the `Elements` present as keys in `structure`. [*4 marks*]

(iii) Implement the instance method `weight()`. Ths should return the sum of the weights of each atom in the molecule. For example, given that the weights of the `Elements` hydrogen and oxygen are 1.01 and 16.0 respectively, and given that there are two atoms of hydrogen to one atom of oxygen in water, the weight of H_2O is $1.01 \times 2 + 16.0 \times 1 = 18.02$ [*8 marks*]

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

- `Element.java`
- `Table.java`
- `Molecule.java`

Final Checklist

Here is a complete list of all the files required for this exam:

```
OneA.java
OneB.java
OneC.java
QuestionOneTester.java
Element.java
Table.java
Molecule.java
```