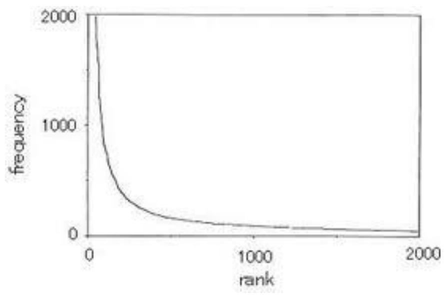
Concordia University
Comp 249 – Fall 2017
Object-Oriented Programming II
Assignment 4 – Due 10:00 AM - Friday December 1, 2017

Important Note:
Since the demos will be held between the 1st and 5th of December, no deadline extension will be granted and no late assignment will be accepted after 10:00 AM (morning) of December 1st .

Purpose: The purpose of this assignment is to allow you practice ArrayList, Interfaces, Generics and Linked Lists.

Part 1: ArrayList & Other subjects

In 1935, the American linguist George Zipf noticed something very peculiar with the books he was reading. Whenever he would count the words in his books, he noticed that most of the words appeared only once and a small number of words appeared very frequently. In addition, this phenomenon seemed to hold true for any text and in any language. Zipf analysed this a little further, and noticed that if you sort the words by frequency and plot the frequency of words versus their rank in the sorted list, you will always get a graph similar to the one below, where a few words have a high frequency (the peak on the left), and most words appear only once (the long tail on the right).



In technical terms:
- Words that appear frequently (the peak on the left) and that happen to be short are called *stop-words*.
- Words that only appear once (the tail on the right) are called *happax legomena*.

Today, this behaviour of language is known as Zipf's law and is used tremendously in the Search Engine industry. In fact, it turns out that Zipf's law is also applicable to many other domains and applications such as monitoring traffic (on the Internet or on the highway), predicting financial activities, and even analysing animal behaviour...

Write a program to verify Zipf's law with your favourite text. Specifically, your program must ask the user for an input text file (please handle potential I/O errors properly), and count how many words the file contains and display the following statistics:

- Display each word in the text along with its rank and frequency.
  You can assume that a word is defined as anything that the method `Scanner.next()` returns and only contains letters. For example "U2", "data-base" and "hi!" should not be counted as words.
  Your program does not have to be case-sensitive. For example, the words "hi" and "Hi" can be considered as same words.
  The list of rank/word/frequency must be displayed in descending order of frequency, and all words with the same frequency must be displayed in alphabetical order (uppercases before lowercases).
- Display the total number of word tokens and word types.
  The number of word tokens refers to the total number of words in the text (the number of times the method `Scanner.next()` returned a `String`); whereas the number of word types refers to the number of different words in the text. For example, if the word "the" appears 30 times, it will count as 30 word tokens, but only 1 word type.
- Display statistics on the *happax legomena*:
  - the number of happax,
  - the percentage of happax (nb happax ÷ nb of word types), and
  - how many of the tokens in the text they account for (nb happax ÷ nb of word tokens).
- Display statistics on the stops words:
  - the number of stop words,
  - the percentage of stop words (nb stop words ÷ nb of word types), and
  - how many of the tokens in the text they account for (total frequency of stop stops ÷ nb of word tokens).

For this assignment, you can assume that a stop word is a word that has a length of 4 characters or less and that appears at least 10 times in the text.

Here is an example of how your program should behave (the file `jokes.txt` is available with this assignment):

```
Enter input file:  jokes.txt

------------------------------------
RANK         FREQ         WORD
------------------------------------
1            17           a
2            10           A
3            7            in
4            7            was
5            5            the
6            5            you
7            4            The
8            4            When
9            4            are
10           4            is
11           4            of
12           4            who
13           4            your
14           3            and
...
34           1            Bakers
35           1            Count
36           1            Did
...
51           1            about
52           1            all
53           1            backward
54           1            baseball
55           1            batteries
...
174          1            why
175          1            will
176          1            wondered
177          1            write
178          1            writes

------------------------------------

Nb of word tokens: 268
Nb of word types: 178

Nb of Happax: 145
% of Happax: 81%
Happax account for: 54% of the text

Nb of stop words: 6
% of stop words: 3%
Stop words account for: 19% of the text
```

Part 2: Linked Lists

Write a program to manipulate Cellular Phones using linked lists.

I) The **`CellPhone`** class has the following attributes: a `serialNum` (long type), a `brand` (String type), a `year` (int type, which indicates manufacturing year) and a `price` (double type). It is assumed that brand name is always recorded as a single word (i.e. Motorola, SonyEricsson, Panasonic, etc.). It is also assumed that all cellular phones follow one system of assigning serial numbers, regardless of their different brands, so no two cell phones may have the same serial number.

You are required to write the implementation of the `CellPhone` class. Beside the usual mutator and accessor methods (i.e. `getPrice(), setYear())` the class must have the following:

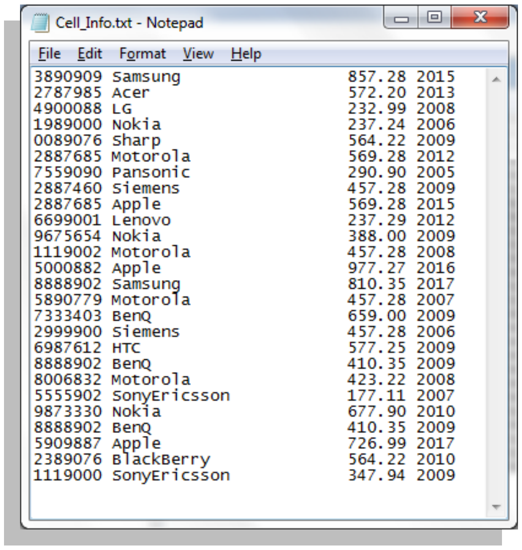
(a) Parameterized constructor that accepts four values and initializes *serialNum*, *brand*, *year* and *price* to these passed values;

(b) Copy constructor, which takes two parameters, a `CellPhone` object and a long value. The newly created object will be assigned all the attributes of the passed object, with the exception of the serial number. *serialNum* is assigned the value passed as the second parameter to the constructor. It is always assumed that this value will correspond to the unique serial number rule;

(c) `clone()` method. This method will prompt the user to enter a new serial number, then creates and returns a clone of the calling object with the exception of the serial number, which is assigned the value entered by the user;

(d) Additionally, the class should have a `toString()` and an `equals()` methods. Two cell phones are equal if they have the same attributes, with the exception of the serial number, which could be different.

II) The file **`Cell_Info.txt`**, which one of its versions is provided with this assignment, has the information of various cell phone objects. The file may have zero or more records. The information stored in this file is always assumed to be correct and following the unique serial number rule. A snapshot of the contents of the `Cell_info.txt` file is shown in Figure 1 below.



```
3890909 Samsung                  857.28 2015
2787985 Acer                     572.20 2013
4900088 LG                       232.99 2008
1989000 Nokia                    237.24 2006
0089076 Sharp                    564.22 2009
2887685 Motorola                 569.28 2012
7559090 Pansonic                 290.90 2005
2887460 Siemens                  457.28 2009
2887685 Apple                    569.28 2015
6699001 Lenovo                   237.29 2012
9675654 Nokia                    388.00 2009
1119002 Motorola                 457.28 2008
5000882 Apple                    977.27 2016
8888902 Samsung                  810.35 2017
5890779 Motorola                 457.28 2007
7333403 BenQ                     659.00 2009
2999900 Siemens                  457.28 2006
6987612 HTC                      577.25 2009
8888902 BenQ                     410.35 2009
8006832 Motorola                 423.22 2008
5555902 SonyEricsson             177.11 2007
9873330 Nokia                    677.90 2010
8888902 BenQ                     410.35 2009
5909887 Apple                    726.99 2017
2389076 BlackBerry               564.22 2010
1119000 SonyEricsson             347.94 2009
```

Figure 1: Cell_info.txt

III) The **`CellList`** class has the following:

(a) An inner class called `CellNode`. This class has the following:

i. Two private attributes: an object of `CellPhone` and a pointer to a `CellNode` object;

ii. A default constructor, which assigns both attributes to `null`;

iii. A parameterized constructor that accepts two parameters, a `CellPhone` object and a `CellNode` object, then initializes the attributes accordingly;

iv. A copy constructor;

v. A `clone()` method;

vi. Other mutator and accessor methods.

(b) A private attribute called `head`, which should point to the first node in this list object;

(c) A private attribute called `size`, which always indicates the current size of the list (how many nodes are in the list);

(d) A default constructor, which creates an empty list;

(e) A copy constructor, which accepts a `CellList` object and creates a copy of it;

(f) A method called `addToStart()`, which accepts one parameter, an object from `CellPhone` class. The method then creates a node with that passed object and inserts this node at the head of the list;

(g) A method called `insertAtIndex()`, which accepts two parameters, an object from `CellPhone` class, and an integer representing an index. If the index is not valid (a valid index must have a value between `0` and `size-1`), the method must throw a `NoSuchElementException` and terminate the program. If the index is valid, then the method creates a node with the passed `CellPhone` object and inserts this node at the given index. The method must properly handle all special cases;

(h) A method called `deleteFromIndex()`, which accepts one integer parameter representing an index. Again, if the index is not valid, the method must throw a `NoSuchElementException` and terminate the program. Otherwise; the node pointed by that index is deleted from the list. The method must properly handle all special cases;

(i) A method called `deleteFromStart()`, which deletes the first node in the list (the one pointed by `head`). All special cases must be properly handled.

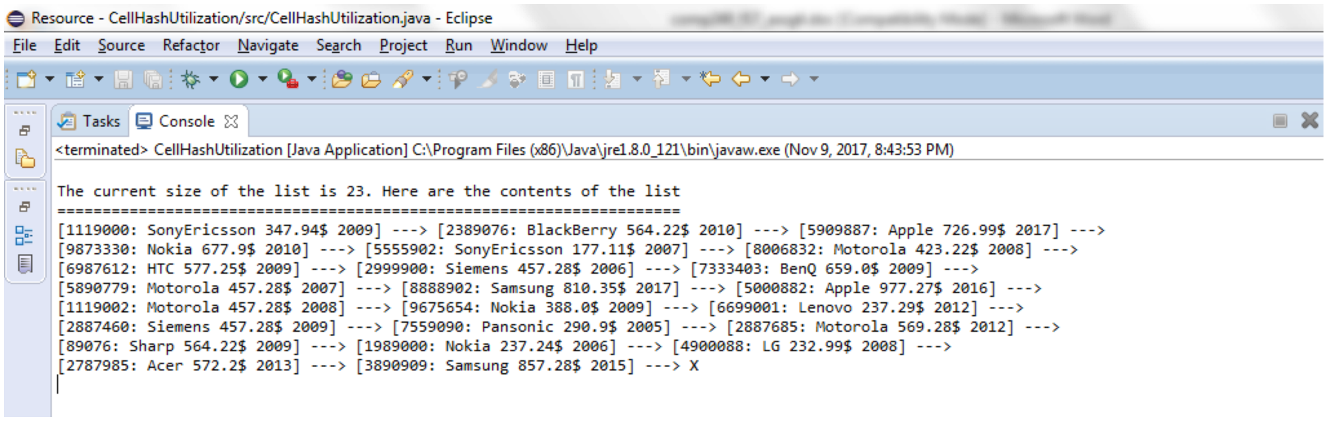
(j) A method called `replaceAtIndex()`, which accepts two parameters, an object from `CellPhone` class, and an integer representing an index. If the index is not valid, the method simply returns; otherwise the object in the node at the passed index is to be replaced by the passed object;

(k) A method called `find()`, which accepts one parameter of type long representing a serial number. The method then searches the list for a node with a cell phone with that serial number. If such an object is found, then the method returns a pointer to that node where the object is found; otherwise, the method returns `null`. The method must keep track of how many iterations were made before the search finally finds the phone or concludes that it is not in the list;

(l) A method called `contains()`, which accepts one parameter of type long representing a serial number. The method returns `true` if the an object with that serial number is in the list; otherwise, the method returns `false`;

(m) A method called `showContents()`, which displays the contents of the list, in a similar fashion to what is shown in Figure 2 below.

(n) A method called `equals()`, which accepts one parameter of type `CellList`. The method returns `true` if the two lists contain similar objects; otherwise the method returns `false`. Recall that two `CellPhone` objects are equal if they have the same values with the exception of the serial number, which can, and actually is expected to be, different.


Resource - CellHashUtilization/src/CellHashUtilization.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Tasks
Console

```
<terminated> CellHashUtilization [Java Application] C:\Program Files (x86)\Java\jre1.8.0_121\bin\javaw.exe (Nov 9, 2017, 8:43:53 PM)

The current size of the list is 23. Here are the contents of the list
=====================================================================
[1119000: SonyEricsson 347.94$ 2009] ---> [2389076: BlackBerry 564.22$ 2010] ---> [5909887: Apple 726.99$ 2017] --->
[9873330: Nokia 677.9$ 2010] ---> [5555902: SonyEricsson 177.11$ 2007] ---> [8006832: Motorola 423.22$ 2008] --->
[6987612: HTC 577.25$ 2009] ---> [2999900: Siemens 457.28$ 2006] ---> [7333403: BenQ 659.0$ 2009] --->
[5890779: Motorola 457.28$ 2007] ---> [8888902: Samsung 810.35$ 2017] ---> [5000882: Apple 977.27$ 2016] --->
[1119002: Motorola 457.28$ 2008] ---> [9675654: Nokia 388.0$ 2009] ---> [6699001: Lenovo 237.29$ 2012] --->
[2887460: Siemens 457.28$ 2009] ---> [7559090: Pansonic 290.9$ 2005] ---> [2887685: Motorola 569.28$ 2012] --->
[89076: Sharp 564.22$ 2009] ---> [1989000: Nokia 237.24$ 2006] ---> [4900088: LG 232.99$ 2008] --->
[2787985: Acer 572.2$ 2013] ---> [3890909: Samsung 857.28$ 2015] ---> X
```

Figure 2: Sample of Displaying the Contents of a CellList

➔ Finally, here are some general rules that you must consider when implementing the above methods:

- Whenever a node is added or deleted, the list size must be adjusted accordingly;
- All special cases must be handled, whether or not the method description explicitly states that;
- All `clone()` and copy constructors must perform a deep copy; no shallow copies are allowed;
- <u>If any of your methods allows a privacy leak, you must clearly place a comment at the beginning of the method 1) indicating that this method may result in a privacy leak 2) explaning the reason behind the privacy leak. Please keep in mind that you are not required to implement these proposals;</u>

IV) Now, you are required to write a public class called **`CellListUtilization`**. In the `main()` method, you must do the following:

(a) Create at least two empty lists from the `CellList` class;

(b) Open the `Cell_Info.txt` file, and read its contents line by line. Use these records to initialize one of the `CellList` objects you created above. You can simply use the `addToStart()` method to insert the read objects into the list. However, the list should not have any duplicate records, so if the input file has duplicate entries, which is the case in the file provided with the assignment for instance, your code must handle this case so thast each record is inserted in the list only once;

(c) Show the contents of the list you just initialized;

(d) Prompt the user to enter a few serial numbers and search the list that you created from the input file for these values. Make sure to display the number of iterations performed;

(e) Following that, you must create enough objects to test each of the constructors/methods of your classes. The details of this part are left as open to you. You can do whatever you wish as long as your methods are being tested including some of the special cases.

Evaluation Criteria (10pts):

Part 1 (5 points)	
Proper Use of ArrayList	1.5 pts
Proper Use of the Comparable Interface	1.5 pts
Simplicity of the Algorithm	1.5 pts
Programming Style	0.5 pts
Part 2 (5 points)	
Design and Corretness of Classes	2.5 pts
Proper and Sufficient Testing of Your Methods	2 pts
Privay Leak Comments and Proposals to Avoid Them	0.5 pts

In addition, please note that part of the submission will involve a short 10 minute demonstration to the marker. No extra preparation is necessary – you only need to explain your code to the marker and answer his/her questions. All members of the group must attend the demo and must be able to explain their program to the marker. Different marks may be assigned to teammates based on this demo. The schedule of the demos will be determined and announced by the markers, and students must contact the marker to reserve their time slot. Demos are mandatory. Failure to do your demo will entail a mark of zero for the assignment. There will be no substitution for a missed demo.

<u>Assignment Submission</u>

- Create **one** zip file, containing the two parts: I, and II separately, where each part contains the files (.java and .html and test cases).
- Naming convention for the zip file:
  - If the work is done by 1 student: Your file should be called *a#_studentID_Part#*, where a# is a3 for assignment3, *studentID* is your student ID number, and *Part#* Part1, or Part2.
  - If the work is done by 2 students: The zip file should be called *a#_studentID1_studentID2*, where a# is a3 for assignment3, *studentID1* and *studentID2* are the student ID numbers of each student, and *Part#* Part1, or Part2.
- Assignments must be submitted in the right folder of the assignments. Upload your zip file at the URL: https://fis.encs.concordia.ca/eas/ as *Programming Assignment* ***4.*** **Assignments uploaded to an incorrect folder will not be marked and result in a zero mark. No resubmissions will be allowed.**