11 Using Java Collections; JUnit; StressTests

In this assignment you should try to write as little code as possible - using the `Java Collections Framework` classes for getting the work done. Also, you should use `JUnit` for all tests.

Portfolio Programs: The Java Collections Framework

Finish all the work for Lab 11. (See below)

11.1 HashMap, JUnit

Finish all parts of Lab 11 and hand in the completed work with your partner.

11.2 Stress Tests

Your job is now to be an algorithm detective. The program we give you allows you to run any of the six different sorting algorithms on data sets of five different sizes using three different `Comparator`s to define the ordering of the data. When you run the program, the time that each of these algorithms took to complete the task is shown in the console.

To run the program you need to do the following:

- Create a new *Java Project* in Eclipse (e.g. *SortingTests*).
- Go to *Preferences* and choose to add a library then choose *Add External jars* to add the file **sorting.jar** to the project.
- Go to the *Run* menu, choose *Run Configurations*, select to make a new configuration. Name it *StressTests* then click on the button to *Select main*. One of the choices should be `sorting.Interactions`. Choose that one. You can now run the program. It will come up with a GUI with several buttons.
- To set up the timing tests you need to go through three steps:

    1. You need to read in the data for the 29470 cities from the file **citydb.txt**. The button *FileInput* opens a file chooser dialog. Select the **citydb.txt** file.

2. Now hit the *TimerInput* button. It lets you select which algorithms to test, which `Comparator`s to use, and what size data should be used in the tests.

   Start with just a few small tests, to see how the program behaves, before you decide to run all tests.

3. Now you can run the actual tests by hitting the **RunTests** button.

   You can repeat the last two steps as many times as you want to.

Run the program a few times with small data sizes, to get familiar with what it can do. Then run experiments and try to answer the following questions:

1. Which algorithms run mostly in quadratic time, i.e. $O(n^2)$?
2. Which algorithms run mostly in $O(n.log_n)$ time?
3. Which algorithms use the functional style, using `Cons` lists?
4. Which algorithm is the *selection sort*?
5. Why is there a difference when the algorithms use a different `Comparator`?
6. Copy the results into a spreadsheet. You may save the result portion in a text editor with a .csv suffix and open it in *Excel* (or some other spreadsheet of your choice). You can now study the data and represent the results as charts. Do so for at least three algorithms, where there is one of each — a quadratic algorithm and a *linear-logarithmic* algorithm.

   Produce a report with a paragraph that explains what you learned, using the *Excel* charts to illustrate this.

11.3 William Shakespeare

The Application

Have you ever wondered about the size of Shakespeare's vocabulary? For this assignment you will write a program that reads its input from a text file and lists the words that occur most frequently, together with a count of how many different words occur in the file. If this program were to run

on a file that contains all of Shakespeare's works, it would tell you the approximate size of his vocabulary, and how often he uses the most common words.

Hamlet, for example, contains about 4542 distinct words, and the word "king" occurs 202 times.

The Problem

Start by downloading the file `HW11.zip` and making an Eclipse project that contains these files. Run the project, to make sure you have all pieces in place. The `Examples` class uses the `tester` package as we have done before.

You are given the file `Hamlet.txt` that contains the entire text of *Hamlet* and a file `InFileReader.java` that contains the code that generates the words from the file `Hamlet.txt` one at a time, via an iterator. Save the file `Hamlet.txt` in the `Eclipse` project directory (where you find the subdirectories `src` and `bin`).

Note: Here you will use the imperative Iterator interface that is a part of Java Standard Library. Make sure to look up the documentation for this interface and understand how it works.

Your tasks are the following:

1. Design the class `Word` to represent one word of Shakespeare's vocabulary, together with its frequency counter. The constructor takes only one `String` (for example the word "king") and starts the counter at one. We consider one `Word` instance to be equal to another, if they represent the same word, regardless of the value of the frequency counter. That means that you have to override the method `equals()` as well as the method `hashCode()`.

2. Design the class that implements the `Comparator` interface, so that the words can be sorted by frequencies. (Be careful!) When you are done, place this class definition as the last part of the class definition of the class `Word`. This is called an *inner class*.

   Note: In this program there will be two ways of comparing the instances of the Word class - by the String that it represents and by the counter for the word that this instance represents.

3. Include in the class `Word` the method that allows you to increment the counter (using mutation), and a method `toString` that prints one line with the word and its frequency.

4. Design the class `WordCounter` that keeps track of all the words we have seen so far. It should include the following methods:

```
// records the Word objects generated by the given Iterator
// for each word record the number of ocurrences
void countGivenWords (Iterator it) { ... }

// How many different Words has this WordCounter recorded?
int words() { ... }

// Prints the n most common words and their frequencies.
void printWords (int n) { ... }
```

   Here are additional details:

5. `countAllWords` consumes an iterator that generates the words and builds the collection of the appropriate `Word` instances, with the correct frequencies.

6. `words` produces the total count of different words that have been consumed.

7. `printWords` consumes an integer `n` and prints the top `n` words with the highest frequencies (using the `toString` method defined in the class `Word`).

Note: The given code expects that you implement the classes as given, with the same names and methods. It will then check whether your program works correctly. That does not mean you do not need to design tests.

Testing of the Shakespeare Project

Of course, you need to test all methods as you are designing them. Design the tests in two stages:

1. For the class `Word` and the the class `WordCounter` use a technique similar to what was done in the past assignments, i.e. design a class `Examples` with the necessary sample data and all tests.

2. Convert all tests into `JUnit` tests. Hand in both versions.

11.4 Documentation

The projects should contain complete `Javadoc` documentation that should produce the documentation pages without warnings. You do not need to submit the documentation pages.