11 Using Java Collections; JUnit

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

Download the file *Words.java* and start a project. The class `Words` contains some `Strings` and `ArrayList<String>` data that we will analyze. Add a class `Algorithms` where you will implement the solutions to the following problems. The tests, of course, are done using *JUnit*.

Problem 1

Design the following two variants of a method in the `Algorithms` class that reverses the order of the words in the given `ArrayList<String>` (we assume each `String` is one word:

1. Design the method `reverseProduce` that takes one argument of the type `ArrayList<String>` and produces a new `ArrayList<String>` that contains all words from the given list, but in a reverse order.

   So, if the original `ArrayList<String>` contained the `String`s `["who" "what" "why"]` the new `ArrayList<String>` will contain the `String`s `["why" "what" "who"]`.

2. Design the method `reverseInSitu` with the `void` return type that mutates the given `ArrayList<String>` by reversing the words it contains.

3. Finally design the method `printWords` that consumes an argument of the type `ArrayList<String>` and prints each entry on a new line, traversing the `ArrayList<String>` using the 'for-each' loop that uses the Java `Iterator`. Use

   ```
   System.out.println(someString);
   ```

   to print each line.

Problem 2: Working with the `StringTokenizer`

Continue working with the same project, designing your solutions in the `Algorithms` class.

1. Look up the `StringTokenizer` class in JavaDocs. The methods there allow you to traverse over a `String` and produce one word at a time delimited by the selected characters. Read the examples. Then write the method `makeWords` that consumes one `String` (that represents a sentence with several words, commas, and other *delimiters* and produces an `ArrayList<String>` of words (`String`s that contain only letters — we ignore the possibility of words like "don't"). The delimiters you should recognize are the comma, the semicolon, and the question mark.

2. The text in the`ArrayList<String> words` in the class `Words` is a secret encoding. It represents verses from a poem - if you read only the first words. Design the method `firstWord` that produces the first word from a given `String`. Use it to decode the poem.

Problems

11.1 Stacks and Queues

The goal of this exercise is to use the *Java* libraries to do the work for us.

1. In looking for a path from one city to another we keep track of the visited cities. For each city we visit, we remember not only the information about that city, but also what city did we come from as we traveled to the newly visited city.

   Use the `HashMap` to keep track of the visited cities. Use the visited city as the `Key` and the city of origin as the `Value`. So, for example, we may have the following information about cities and traveling between them:

```
Boston, MA - visited first: came from 'null'
Albany, NY - we came from Boston, MA
Concord, NH - we came from Boston, MA
Montpellier, VT - we came from Concord, NH
Trenton, NJ - we came from NY
Harrisburg, PA - we came from Trenton, NJ
```

Define the class `Path` that records this information about the `City` data used in the Lab 11. **Use `HashMap` from the** *Java Collections Framework***.** Make sure you include the above example in your tests - getting all the information about for these cities by reading the file *caps.txt* that has the data for the capitals of the 48 congruent US states. Use the file *InFileCityTraversal* to read in the file - save the data to an `ArrayList`.

2. Define in the class `Path` the method `pathTo` that produces an `ArrayList` of `City`-s we need to go through to get to the given `City`. So, for the above example, we would expect the following results:

```
pathTo(Boston, MA) --> [Boston, MA]
pathTo(Albany, NY) --> [Boston, MA; Albany, NY]
pathTo(Harrisburg, PA) --> [Boston, MA;
                                Albany, NY;
                                  Trenton, NJ;
                                    Harrisburg, PA]
```

3. Define in the class `Path` the method `contains` that determines whether the given `City` is in this `Path`.

4. Define the method `directionsFromTo` that consumes the city of origin and our desired destination and produces the travel directions as a `String`. For example,

```
directionsFromTo(Boston, MA : Boston, MA) produces:
"Start in Boston, MA
End in Boston, MA"

directionsFromTo(Boston, MA : Harrisburg, PA) produces:
"Start in Boston, MA
Boston, MA to Albany, NY
Albany, NY to Trenton, NJ
Trenton, NJ to Harrisburg, PA
End in Harrisburg, PA"
```

****

We now want to keep track of the neighbors of the cities we visited (and we plan to visit soon) (the `ToDo` checklist). So, for example, if we visit `Boston, MA`, we will add to the `ToDo` checklist all of its

neighbors. However, there are some restrictions. We do not add a neighbor to the checklist if it is already in the `Path`.

The interface `ToDo` describes the desired behavior:

```
interface ToDo{
// add a new neighbor to the ToDo checklist
// unless it already appears in the given Path
public void add(City city, Path path);

// remove the given city from the ToDo checklist
// return false if the city is not in the checklist
public boolean remove(City city);
}
```

5. Define the class `ToDoStack` that keeps track of the neighbors to visit soon that uses the *Java* `Stack` class to implement the `ToDo` interface as a *stack*.

6. Define the class `ToDoQueue` that keeps track of the neighbors to visit soon that uses the *Java* `LinkedList` class to implement the `ToDo` interface as a *queue*.

Testing of the Stacks and Queues Problem

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

1. First design the tests using the *Examples* class and the *tester* package as we have done before.

2. Now convert all tests into *JUnit* tests. Hand in both versions.

11.2 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.

11.3 Documentation

Both of the projects should contain complete *Javadoc* documentation that should produce the documentation pages without warnings. You do not need to submit the documentation pages.