11 Using Java Collections; JUnit

Etude: Eliza Doolittle

In the lab you started working on the *Eliza* program that allows the computer to interact with the user by providing replies to a series of questions. As the etude for this assignment, finish the program. Include in your portfolio a sample transcript of the user-computer interaction.

Preamble

Goals

The first part of this assignment consists of a small program that uses interfaces and classes either from Java's standard libraries, or from our earlier labs and assignments. The goal is to give you a bit of design freedom: You get to decide which parts of the standard libraries, or which classes and interfaces we already designed are the most suitable to use. If you design well, this assignment should be fairly straightforward.

The goal of the second part is to give you a practice in designing reusable library-style classes using the Java program design standards for design, documentation and also for testing. The program you produce will eventually use the JUnit test tools and will include documentation in the style that allows you to produce *Javadoc* documentation for your program.

Hints

Some or all of the following interfaces and classes are likely to prove useful. In the *java.lang* package: *Comparable*, *Iterator*, *List*, *Map*, *Set*, *Collections*.

11.1 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 **HW11** that contains these files. Add **jpt.jar** as a *Variable* to your project. Run the project, to make sure you have all pieces in place. The main method is in the class *Examples*.

You are given the file **test.txt** that contains the entire text of *Hamlet* and a file **Week11.java** that contains the code that generates the words from the file **test.txt** one at a time, via an iterator.

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.

The classes *Tester* and *Examples* contain a test harness similar to the *SimpleTestHarness* used in the previous two assignments, but improved to catch exceptions raised while running the tests. More about this later...

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*.

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.
   void countWords (*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. *countWords* 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*).

Part 2: The Testing

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

1. For the class *Word* use a technique similar to what was done in the past two assignments, i.e. design a class *SimpleTests* that instantiates the class *Tester* as well as the necessary sample data and collects all tests in a method *void run*(). At the end of this method it invokes either the *testReport* or the *fullTestReport* method to report on the results.

2. When designing the class *WordCounter*, upgrade to the next level of the test harness. The class *Tester* contains the following driver for the tests:

```
// run the tests, accept the class to be tested as a visitor
   void runTests(Testable f) {
      this.n = 0;
      try {
         f.tests(this);
      }
      catch (Throwable e) {  // catch all exceptions
         this.errors = this.errors + 1;
         console.out.println("Threw exception during test " + this.n);
         console.out.println(e);
      }
      finally {
         done();
      }
}


      // to be run after all tests have been performed
      public void done(){
         if (this.errors > 0)
            console.out.print("Failed " + this.errors + " out of ");
         else
            console.out.print("Passed all ");
            console.out.println (this.n + " tests.");
      }
```

The class *Examples* implements the *Testable* interface that contains just one method:

```
void tests(Tester t);
```

Inside of this method the class *Examples* invokes the appropriate *test* methods on the instance *t* of the *Tester*.

So we have a chicken and egg problem here. The class *Tester* wants to know what is the *Examples* instance that is running the tests, so that it can invoke the method *tests*(*Tester t*) defined in the *Examples* class inside of the *Tester*'s *try* clause.

The class *Examples* in turn needs an instance of the class *Tester* so that it can invoke each *test* method inside of the method *tests*(*Tester t*).

The main gain is that every invocation of the methods *test* is wrapped inside of the *try* clause and if an exception is thrown, the error report indicates which one of the tests failed.

The only thing you need to do is to include all your tests and the needed sample data inside of the *tests*(*Tester t*) method in the class *Examples*.

This prepares us for the third way of running tests, namely using **JUnit** - Java's standard test framework.

3. **Introducing JUnit:** *Do this for a practice.* Then use JUnit for tests in the Part 3 of this assignment.

   You will now rewrite all your tests using the JUnit. In the **File** menu select **New** then **JUnitTestCase**. When the wizard comes up, select to include the *main* method, the constructor, and the *setup* method. The tests for each of the methods will then become one test case similar to this one:

```
/**
 * Testing the method toString
 */
public void testToString(){
      assertEquals("Hello: 1\n", this.hello1.toString());
      assertEquals("Hello: 3\n", this.hello3.toString());
}
```

   We see that *assertEquals* is basically the same as the *test* methods for our test harnesses, they just don't include the name of the test. Try to see what happens when some of the tests fail, when a test throws an exception, and finally, make sure that at the end all tests succeed.

11.2 Stacks, Queues, and Priority Queues

In our next assignment we will need to keep track of accumulated values — places we should visit next. However, the way how we add/remove items from this accumulator will depend on our choice of algorithms. Therefore, we start with a common interface, and design three different implementations of this interface.

The *Accumulator* interface is defined as follows:

```
/**
  * <P>An interface that represents a container for accumulated collection of
  * data elements. The implementation specifies the desired add and remove
  * behavior.</P>
  * <P>The expected implementations are Stack, Queue, and Priority Queue.</P>
  */
public interface Accumulator<T>{


    /**
      * Does this <CODE>{@link Accumulator}</CODE> contain any data elements?
      * @return true is there are no elements in this
      * <CODE>{@link Accumulator}</CODE>.
      */
    public boolean isEmpty();


    /**
      * Change the state of this <CODE>{@link Accumulator}</CODE> by adding
      * the given element to this <CODE>{@link Accumulator}</CODE>.
      * @param t the given element
      */
    public void add(T t);


    /**
      * Change the state of this <CODE>{@link Accumulator}</CODE> by removing
      * the given element to this <CODE>{@link Accumulator}</CODE>.
      * Produce the removed element.
      * @return the removed element
      */
    public T remove();
}
```

1. Design the class *MyStack<T>* that implements the *Accumulator<T>* interface by always removing the most recently added element.

2. Design the class *MyQueue<T>* that that implements the *Accumulator<T>* interface by always removing the least recently added element.

3. Design the class *MyPriorityQueue<T>* that contains an instance of a *Comparator<T>* and implements the *Accumulator<T>* interface by always removing the element that has the highest priority as determined by its *Comparator<T>*.

4. Use the *JUnit* for all tests for these classes.

Note: You can decide on your own what will be the class of data that will provide the elements to use in testing these classes.

The Documentation: a concise summary

You may have noticed that the style in which we write documentation for this assignment has changed. When written in the well formatted *javadoc* style, the comments can used to generate web pages of documentation with cross-references and browsing capabilities. There are a few basic rules, the rest you should learn on your own, gradually, as you become more and more skilled Java programmers.

Here are comments to specify the name of the file, and the class definition:

```
/*
 * @(#)Word.java   17 November 2006
 *
 */


/**
 *
 *  <P><CODE>Word</CODE> represents one word and its
 * number of occurrences counted in the
 * <CODE>{@link WordCounter WordCounter}</CODE> class.</P>
 *
 * @see Comparable
 *
 * @author Viera K. Proulx
 */
public class Word implements Comparable {
```

The *@author* and *@see* identify the author and provide a cross-reference to other classes as specified.

Each field in the class has its own comment:

```
/**
 * the frequency counter
 */
public int counter;
```

Each method has a comment that includes a separate line for each parameter as well as for the return value:

```
/**
 * Compare two <CODE>Object</CODE>s for equality
 *
 * @param obj the object to compare to
 * @return true if the two objects have the same contents
 */
public boolean equals(Object obj){
```

The *@param* has to be followed by the identifier used for that parameter. The <CODE> and < /CODE> tags specify the formatting for the document to be the teletype font for representing the code.

Eclipse helps you to write the documentation. If you start the comment line with /** and hit the return, the beginnings of remaining comment lines are generated automatically, and you only need to add the relevant information.

When you have finished all the documentation, select the item **Generate Javadoc...** in the **Project** menu. To see your web pages, just open the tab *doc* in the **Package Explorer** window under your project and double click on the *index.html*.