COSC102 **Spring 2015**

DS 1: Word Generator — Homework 5

Due date: **Monday, March 30th, at 11:55 p.m. on Moodle**

The objectives of this homework are

- to gain experience using existing structures to build more complex ones and
- to improve your class design skills.

The design and implementation of the word generator program described below use vectors and associations. You are required to use the `Vector` and `Association` implementations, which we studied in class and used in the lab. Thus, you have to link your program to the `bailey.jar` as in the lab, so to have access to the `structure5 package`, which contains the appropriate classes.

Problem Description

This assignment comes from the world of artificial intelligence. The idea is to read a collection of text and then use some properties of that text to generate some new, "machine generated" text. The results can make interesting reading, but you probably don't want to use this to generate your course term paper.

The method for generating the text uses simple probability to "train" our program to generate new words based on an input text. To do this, we read the text character by character and keep track of how often each three-character sequence appears. From this, we can compute the probability that a certain character will immediately follow two given characters.

For example, if the text is `the theater is their thing` we have appearing after `th`: `e` three times, `i` once and no other letters.

So the probability that `e` follows `th` is .75; the probability that `i` follows `th` is .25; the probability that any other letter follows `th` is 0. The input text has to be processed and stored in a structure such that the probabilities are available.

To machine generate the output text, two letters (for example, the first two in the input text, or two random characters) are picked to use as a beginning. Then subsequent characters are based on the preceding two characters and the probability information gathered from the input text.

For example, consider the input text above, and choosing `th` as the starting string. With a probability of .75, we will choose `e` as the next letter. So if the text becomes `the`, what are the possibilities to follow `he`? A space, an `a`, or an `i` can follow with equal probability. If `a` is chosen, the text is `thea` and the process continues, choosing among the possible letters to follow `ea`.

The program stops when we encounter a 2-character sequence for which there is no subsequent character (which will occur only if the last two characters of the input occur only in that location) or when we have generated a total number of characters as specified as a parameter to the program.

Design

You should think about the design of this program carefully before beginning any implementation. A good choice of a data structure is essential to solve the word generation problem reasonably easily and efficiently. I encourage you to write a design document, which you should bring when you seek help to get feedback on your plan (more information below).

A table constitutes a good data structure to solve the word generator problem. The primary functionality of your program is

1. to update the probabilities in a table, given a new triple of characters, and
2. to select a next character, given a pair of characters and the probabilities stored in the table.

You should develop a `Table` class which stores in a `Vector` of `Association`s mappings between 2-character sequences with possible subsequent characters. Each `Association` should have a 2-character pair (stored as a `String`) as its key, and a `FrequencyList` as its value. (The key length could be a constant so you can easily modify your program later to generate less gibberish text.)

Each frequency list should be an object of another class, `FrequencyList`, that you will define. It should keep track of each character that appeared after the given 2-character pair, along with the number of times it appeared. There are many ways to implement the `FrequencyList`. A good possibility is another `Vector` of `Association`s. In this case, the `Association`s have a key of a single character (which could be stored as a `String` for consistency with the rest) and a value which is the count (stored as an `Integer`) of the number of times that letter occurred after the 2-character pair with which this `FrequencyList` is associated. Think carefully about what methods the frequency list needs to support and which other instance variables might be useful. The data structure design built from these two classes has the benefit of having only as many entries as necessary for the given input text.

Your main method for the program should be written in a client class, `WordGen`, which reads the input text from a file. The client class used to the two data structure classes you are writing to build the table, print it and finally print out a randomly-generated string based on the character sequence probabilities from the input text. All I/O (input and output) happens in the `main` method of the `WordGen` class.

Input

First develop and debug your program using as input a `String` constant (e.g., `the theater is their thing`) until it works properly. Print the created table in a format that facilitates to check the correctness of your program. That is what I call the **first output of the table** in the grading scheme and is due on **Friday, 27th of March at 5:00 P.M.**, as a post on the dedicated *A5 Q/A Forum: Table output.*

After your program can build a table for a simple input change it to accept three command-line arguments.

- The first argument is a filename from which to read the input text.
- The second one is the two letters used to start the generated text.
- The third is an integer used to stop the generated text.

When arguments are not provided (none are or the latter ones are missing), your program should use default values. Information to read from a file is provided below. Feel free to try the provided text file, e.g. `whosonfirst.txt` and other text of your choice. The file has to be in plain text, or format characters may cause issues. Since your `readme` has to contain the tests you used, document the testing you do as you develop your program. Another “interesting” input to try out is the Java source code for one of the classes you have implemented.

Output

You should output the table in an easy to follow format, followed by the machine generated text corresponding to the table frequencies.

The generated text starts with the pair of letters provided as command-line argument or with a random pair if not specified. As stated above, you should generate text until there is either no next character available or until you have produced the maximum number of characters, that is specified by the command-line parameter or set by default when none is provided.

Setup

- Create a new directory in your `102/hw` folder
- Make a `New Java class...` named `WordGen`.
- Check in `Preferences` that the link to the external jar file, `bailey.jar`, still exists.

Try a small program that uses the `Vector` Bailey's structure as the one you wrote in the lab. Make sure you include the following lines

```
import structure5.Vector;
import structure5.Association;
```

and do not include ~~`import java.util.*;`~~ in any of your classes or there will be a conflict between `Java API Vector class` and the `bailey.jar` one. Instead import classes from `java.util` using their full names. For example use

```
import java.util.Random;
import java.util.Scanner;
```

Design Document

Read this document all the way through at least **three** times while working on your design document.

First draw pictures to illustrate how an example string is sequentially processed to create the table data. The design document should present the general approach you will take to the problem and specifically

- include descriptions of the classes, instance variables and important instance methods,
- briefly describe pseudo-code for the main functionality of your program (the items 1. and 2. listed above) and
- describe simple data and tests you are planning to use to check the correctness of the program as you gradually develop it.

For the different data representation think about where **uniqueness** is fundamental and document that information.

File

You may have used a `Scanner` object to read input from the console

```
Scanner scan = new Scanner(System.in);
```

The actual parameter `System.in` represents the standard input stream, normally the keyboard. To use a file for input instead, you can initialize a `Scanner` object in the following way:

```
Scanner scan = new Scanner(new File(filename));
```

where `filename` is a string representing the location/name of the file to open. The `new File()` part will throw an exception if the file doesn't exist, so you have to surround it with a `try-catch` as shown next.

```
//required at the top of the file
import java.io.*;

//in main method
Scanner sc = new Scanner("the theatre is the thing");
if (args.length > 0) {
  try {
     // read from filename provided by command-line argument
     sc = new Scanner(new File(args[0]));
  } catch (FileNotFoundException e) {
     System.err.println("error: file does not exist");
     return;
  }
}
```

When working with files you are forced by Java to acknowledge that error situations—out of your control—can occur. You should read about Java Exception Handling to learn more.

Notes

- Close resources when you're done using them: most I/O objects (the `Scanner`) have `close()` for this.
- When debugging the table construction, keep in mind that you can use the built-in `toString` methods in `Vector` and `Association` very easily, but they may not present the information in a convenient form. Write custom `toString` methods for your `FrequencyList` and `Table` classes.
- Treat spaces and punctuation just like any other characters. The word generator program should be case-sensitive.
- You might find the word frequency program on page 48 of the Java Structures book helpful.

Grading

For this homework you are required to write **Javadoc** style comments and to submit their associated **html files**. (It might be worth it to write your design document as Java comments and clean up the writing as you develop the actual code.) To produce Javadoc you have to include comments following the specific Javadoc style (see textbook and distributed code) and use a tool to automatically generate the API webpages: this feature is available in `DrJava` under the `Tools` menu. Make sure for each class to include a general description at the top of the file, with authorship and date information.

The `readme` should include

- your student information at the top,
- sufficient examples to demonstrate how you tested your program. For some input and generated output—short and long ones (long ones can be truncated or submitted as separated files included in the submission)—write up a paragraph with the observations you made about the generation correctness.

If you have not fully implemented the functionality of the word generator, list in your `readme` the parts that work (and how to test them) and the ones you attempted so as to receive partial credits.

The program design grade will be based on the appropriateness of your choice of internal data structures and methods. The program style especially matters in this homework: good formatting, Javadoc and naming conventions are expected.

Your grade will based on the following weights:

First Output of Table due March 27th 5:00 P.M.	10
Command line argument & file handling	10
Table	30
Generated text	20
readme with tests	10
Program design and Javadoc	20

Credit This assignment was created by James Teresco.