Text Generator

Assignment 2
Due Sunday, February 10, 2013
CSC 062: Spring, 2013

Goals

The goals for this assignment are for you to become proficient in using objects to build more complex structures, to learn the importance of careful class design, to learn the ArrayList and Association, and to use generics properly.

`ArrayList` is part of java.util, with documentation found in the Java 7 Javadoc pages linked to from the class "documentation" page. `Association` is part of Bailey's structure5 package. Javadoc for that is also available from a different link in the class "documentation" page.

Warning: This assignment is significantly harder than the last one and will take a lot of time. Please start early to save yourself lots of grief. Also, it is strongly recommended that you spend significant time thinking about your design before you start coding.

Text Generator

In this assignment we will try an experiment in Artificial Intelligence. The idea is to write a program which will read in text and then use that text to generate some new text. The method for generating the text uses simple probability - we read the text word by word, counting punctuation symbols as words. We keep track of how often each three-word sequence (trigram) appears. From this we can compute the probability that one word will immediately follow two given words. For example, consider the text that includes the following lines from Rudyard Kipling's poem "If":

> If you can keep your head when all about you
> Are losing theirs and blaming it on you,
> If you can trust yourself when all men doubt you,
> But make allowance for their doubting too;
>
> If you can wait and not be tired by waiting,
> ...

In the text shown, "can" occurs three times after "If you" and no other word follows "If you". On the other hand the words "keep", "trust", and "wait" each occur once after "you can". So the probability that *can* follows *If you* is 1; the probability that anything else follows *If you* is 0. The probability that each of *keep*, *trust*, and *wait* follows *you can* is .33; the probability that any other word follows it is 0.

Once we have the text processed, and stored in a structure that allows us to check probabilities, we then pick two words (for example, the first two in the input text) to use as a beginning for our new text. Finally we use random numbers to choose subsequent words based on the preceding two words and the probability information. Limit the output text to no more than 400 "words" (counting punctuation as words).

Warning: If you start with the first two words of the input text, then virtually always you will find one or more elements in the frequency list. The only exception occurs if you generate the last two words of the input. If those last two words did not occur earlier in the input, you won't find the pair in the table. If that happens, you may just terminate generating words (e.g., by quitting or just returning empty strings from the method). If you want to be more creative, randomly pick some pair in the table and start generating again from there.

Important Considerations

You should think about the design of this program carefully *before* sitting down at a computer. What would constitute a good data structure for this problem? The table of information should support requests of the form:

- update frequency lists given a new triple of words.

- select a new word given a pair of words and a random number (double) between 0.0 and 1.0. The probability of selcting the new word should be based on the probability that the word follows the pair in the input text.

A 3-dimensional array might seem reasonable at first, but its size would be unbelievably large. Even if we limited ourselves to 1000 words, there would be one billion possible triples. Instead I would like you to create a `TextGenerator` class which is implemented as an `ArrayList` of `Associations`. Each `Association` would have a 2-word pair as its key, along with a value which is a frequency list. The frequency list would keep track of which words appeared after the given 2-word pair, along with the number of times each word has appeared.

Each frequency list should be an object of a class called `FrequencyList` that you will define. A `FrequencyList` will also be implemented as an `ArrayList` of `Associations`, except that its key will be a single word, and its value will be a count of the number of times that word appears after its associated two-word pair. Think carefully about what methods the frequency list needs to support and any other instance variables that might be useful.

The data structure built from these two classes has the benefit of having only as many entries as necessary for the given input text (i.e., if two words never appear adjacent to each other in the text, there will be no entry for the pair.)

Warning: As a general rule it is good to be as specific as possible with import statements. Thus, when you import the package with the class Random, use

```
import java.util.Random;
import java.util.ArrayList;
import structure5.Association;
```

If your program included "`import java.util.*;`" along with "`import structure5.*` the program might get confused as to which version of classes it should use as there are several classes in java.util that have the same names as those in structure5. It is best to be safe even when we don't have conflicts.

Getting Started

We want to encourage you to start using Java packages, so this week's starter files are setup for you. To begin your lab, create a new Java project called Assignment2 in Eclipse. Don't forget to add the Bailey variable to the project. Next copy the "wordsGeneric" subfolder of this week's assignment folder into your project directory's `src` directory. Assuming you followed last week's lab instructions to create your workspace, you should be able to use the following command:

```
cp -r /common/cs/cs062/assignments/assignment2/wordsGeneric ~/mycs062/Assignment2/src/
```

You should also copy all of the text files from the assignment directory to some location that is convenient and easy to access. These are sample input files that you can use to test your program. I've tried to pick files with sufficient repetition of triples that something interesting will happen.

Input and output

I have provided you with startup code in the main method of class `TextGenerator` that will pop up a dialog box to allow the user to choose a file containing the input text. Notice that the window it pops up will show files from your home directory (this shows up in the left margin of finder windows with a house icon and your login name next to it). You may find it helpful to place your input files in that directory (or perhaps a subdirectory of that directory) so that you won't take too much time to find them.

Also in the main method is code that will read the entire file chosen, break it down into separate words and punctuation, and make them available in an object associated with identifier `ws` of type `WordStream`. You can get successive words from `ws` by executing `ws.nextToken()` repeatedly. Before getting a new word, always check that there is one available by evaluating `ws.hasMoreTokens()`, which will return `true` if there are more words available. If you call `nextToken()` when there are no more words available (i.e., you've exhausted the input), then it will throw an `IndexOutOfBoundsException`.

After the input has been processed you should generate new text using the frequencies in the table. You may start with a fixed pair of words that appears in the table (e.g., the first pair) or choose one randomly. Generate and print a string of at least 400 words so that we can see how your program works. When printing the text, please generate a new line after every 20 words so that you don't just generate one very long, unreadable, line.

Finally, we'd like you to print the table of frequencies so we can check to see if your table is correct on our test input. Here are some lines of output based on the lyrics for Bob Dylan's "Blowin' in the wind" to use as a guide:

```
The table size is 122
<Association: <how,many>=Frequency List: <Association: roads=1><Association: seas=1>
    <Association: times=3><Association: years=2><Association: ears=1><Association: deaths=1>>
<Association: <many,roads>=Frequency List: <Association: must=1>>
<Association: <roads,must>=Frequency List: <Association: a=1>>
<Association: <must,a>=Frequency List: <Association: man=2><Association: white=1>>
...
```

Note that I cheated and manually wrapped the first line of output so that it is readable. Your output need not wrap. Note that most of this output comes from the `toString` *methods of* `Association` *and* `ArrayList`*, though I did some extra work to print each line of the table on a separate line.*

This output indicates that after the word pair, "how many", the words "roads", "seas", "times", "years", "ears", and "deaths" each occurred once. Whereas after "many roads", only "must" occurred and it only occurred once.

Helping Classes

Because we want you to focus your attention on the use of `ArrayList` and `Association` library classes, we've provided you with a few other classes to make your lives a bit easier. They are:

WordStream Described above, it processes a text file to break up the input into separate words that can be requested using method `nextToken`.

StringPair A class that provides objects that consist of a pair of strings. These should be used as the keys to your table.

Pair This is a generic class that `StringPair` extends by specializing the type variables to both be `Strings`. You should not directly use this class in your program. Instead use the more compact name `StringPair` from the more specialized class defined above. We provided this class only to show how easy it is to define a generic class that can be instantiated in many ways.

There is one piece of information about the `Association` class that we did not highlight in class that you will likely find very useful. The `equals` method in Association only compares key values. That is if two objects from class `Association` are "equal" according to the method, then their keys are the same, but their values may be different. Thus, to find out if an association with a given key is in an arraylist (and where it is), just create a new association with that key and use the `indexOf` method to determine its index in the list. If -1 is returned, then it is not there, whereas if a non-negative integer is returned then you get the index of an association with that key in the arraylist.

Program Design

We suggest that you write a frequency list class first. It can be implemented as an ArrayList of Associations, where each Association holds a word and the number of times it occurs. When a word is added, if it already occurs in the list then bump its associated count by 1. If it doesn't exist, add it with count 1. Also include a method `get(double p)` that returns a word based on a probability p.

It's probably best to illustrate this with an example. Suppose the word frequency list includes "the" with frequency 1, "a" with frequency 2, and "some" with frequency 1. Adding all the frequencies together we get a total of 4. We return "the" 1/4 of the time (e.g., whenever $0 \leq p \leq .25$), "a" 2/4 of the time (e.g.,

whenever .25 < p ≤ .75) and "some" 1/4 of the time, (e.g., whenever .75 < p ≤ 1). Write this class and test it thoroughly by adding a main method to make sure all methods work correctly. I suggest printing out a representation of the frequency list first to make sure that the table is correct before attempting to write or test the probabilistic `get` method.

Once the frequency list is ready, prepare the class to build the table holding pairs of words and frequency lists for words that follow that pair. That is, elements of the table are of the form

$$[< \texttt{Word1}, \texttt{Word2} > \rightarrow < w_1, n_1 >, \ldots, < w_k, n_k >]$$

where this notation represents an association where the pair $< \texttt{Word1}, \texttt{Word2} >$ is the key, and $< w_1, n_1 >, \ldots, < w_k, n_k >$ is a frequency list that is the value of the association. That is, the triplet `Word1 Word2` w_1 occurs n_1 times in the text, and the triplet `Word1 Word2` w_k occurs n_k times in the text.

Thus the List representing the table has interface `List<Association<StringPair, FreqList>>` and is constructed from a constructor call like `new ArrayList<Association<StringPair, FreqList>>(size)`, assuming you have named your frequency list class `FreqList`.

Grading

You will be graded based on the following criteria:

criterion	points
reads empty input file without error/crash	2
reads and stores non-empty input file correctly	2
generates correct frequency table output	3
generates "correct" new text	3
general correctness	1
appropriate comments (including JavaDoc)	2
style and formatting	2
submitted correctly	1

Extras

There are many ways in which you might extend such a program. For example, as described, word pairs which never appeared in the input will never appear in the output. Is there a way you could introduce a bit of "mutation" to allow new pairs to appear?

These "trigrams" of words are used in natural language processing in order to categorize kinds of articles and their authors. Think about how you might use tables like those given here to help determine if two articles are written by the same author.

What to hand in

As usual, export the entire folder from Eclipse to your desktop, change the name of the folder to "Lastname-Assignment2" (where you need to replace *Lastname* by your last name). Make sure the folder contains both your .java and .class files. Then drag it into the dropoff folder in `/common/cs/cs062/dropbox`, where an alias for the cs062 folder should be sitting on your desktop already.

As always, test your classes thoroughly before turning in your project. Feel free to use the text files in the assignment folder to test your program. While you may not compare code with other students, you may compare the contents of the tables you generate.