Text Generator

Due Sunday February 9, 2014

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, and to gain experience using `ArrayList`, `Association`, and generics.

`ArrayList` is part of java.util, with documentation found in the Java 6 Javadoc pages linked to from the class "handout" page. `Association` is part of Bailey's structure5 package. Javadoc for that is also available from a different link in the class "handout" page.

The files for this assignment can be found at `/common/cs/cs062/assignments/assignment02`. This directory contains some sample pieces of literature (e.g. Martin Luther King's "I have a dream" speech) as well as a directory entitled `wordsGeneric`. This directory is the package containing the Java starter code for this project.

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!

Text Generator

In this assignment, we will use a basic technique in Artificial Intelligence to automatically generate text. You will write a program that reads in text and then uses that text as a basis to generate new text. The method for generating new text uses simple probability. First, we read in a piece of text word by word (we consider punctuation symbols to be words) keeping track of how often each three-word sequence (trigram) appears. For example, consider the following excerpt 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 this excerpt, the trigrams are: "if you can", "you can keep", "can keep your", "keep your head", "your head when", "head when all", etc.

Once we have counted all of the trigrams, we can compute the probability that one word will immediately follow two other words. For example, the word "can" occurs three times after the words "if you". Every time the words "if you" appear, they are always followed by the words "can". Thus, the probability that "can" follows "if you" is 1.

$$p(can|if\ you) = 1$$

I.e., the probability of observing the word "can" given that we observed the words "if you" is 1. The probability that any other word comes after "if you" is 0. Consider another example. The words "you can" appear 3 times: followed by "keep", "trust", and "wait" respectively. Thus,

$$p(keep|you\ can) = 1/3$$
$$p(trust|you\ can) = 1/3$$
$$p(wait|you\ can) = 1/3$$

Again, the probability that any other word besides "keep", "trust", or "wait" appears after "you can" is zero.

Once we have the text processed, and stored in a data structure that allows us to compute probabilities, we then pick two words (for example, the first two in the input text) to use as the beginning for our new generated text. Given our two beginning words, we use a random number generator to choose the subsequent words in our text based on the preceding two words and the probabilities. Limit the output text to no more than 400 "words" (including punctuation).

Note: There may be two words that were seen in the text but were never followed by any word. For example, in the excerpt of the Kipling poem above, the words "by waiting" occur once but are never followed by a word (since they are the last two words in the excerpt). In this case, you can either stop generating words or feel free to think up a more creative solution.

Program Design

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? Your data structure should support requests of the form:

- Update data structure 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 selecting 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 and a frequency list as its value. The frequency list would keep track of those words that appeared after the given 2-word pair along with their frequencies. For example, in the `TextGenerator` class we would have an `ArrayList` that would contain an `Association` whose key was the 2-word pair "you can" and whose value was a list that contains the following words and frequencies: {"keep", 1}, {"trust", 1}, {"wait, 1}.

The frequency list is a class you will define. How should the frequency list be implemented? Well... another `ArrayList` of `Associations` sounds like a good idea! Thus, the class `FreqList` should contain an `ArrayList` instance variable which consists of `Associations` in which the key is a single word and the value is the frequency of that word (i.e. the number of times that word occurs after the pair with which the `FreqList` is associated). Think carefully about what methods the `FreqList` needs to support and any other instance variables that 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 (i.e., if two words never appear adjacent to each other in the text then there will be no entry for the pair.)

`FreqList` class

We suggest that you write the `FreqList` class first. `FreqList` should contain an array list 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 array list then its value (i.e. its frequency) is incremented by 1. If it doesn't exist, add the word to the array list with a value (i.e. frequency) of 1.

Also, include a method `get(double p)` that, given a probability p, returns a word from the array list. In our example above, the `FreqList` for the 2-word pair "you can" contains {"keep", 1}, {"trust", 1}, {"wait, 1}. Adding all of the frequencies together we get a total of 3. Thus, given a probability p ($0 \leq p \leq 1$), we return the word "keep" whenever $0 \leq p < 1/3$, we return the word "trust" whenever $1/3 \leq p < 2/3$ and we return the word "wait" whenever $2/3 \leq p \leq 1$.

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.

`TextGenerator` class

Once the `FreqList` class is complete, write the `TextGenerator` class which contains an `ArrayList` of `Assocations` whose keys are 2-word pairs and values are `FreqLists`. We have included a class `StringPair` that can be used to represent the 2-word pairs, i.e. the `StringPair` class can be used as the key. Thus, the array list in `TextGenerator` should have interface `List<Assocation<StringPair, FreqList>>`.

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.

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 below, 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 assocations.

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.

Input and output

We 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.

We have included some text files (ending with suffix "txt" in the assignment folder that you can use to test your program. We've tried to pick files with sufficient repetition of triples that something interesting will happen.

Also in the main method is code that will read the entire file chosen, break it down into separator 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 or choose words 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 in some reasonable fashion 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 Donovan's "Blowin' in the wind":

```
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.

Grading

You will be graded based on the following criteria:

criterion	points
functionality/correctness	12
appropriate comments (including JavaDoc)	3
appropriate use of data structures	2
style and formatting	2
submitted correctly	1
extra credit	1

NOTE: Code that does not compile will not be accepted! Make sure that your code compiles before submitting it.

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 **Assignment2_LastNameFirstName**
- Make sure the folder contains both your .java and .class files
- Drag your folder into the drop off folder

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.