Now and again you may find yourself with a terrible need for a nonsense word, or passage of text. Certainly it seems that a lot of the people sending me spam emails have such a need. A quick google search will find plenty of web pages that may more or less meet your needs, and if you want something more sophisticated you can even find sites that generate nonsense scientific papers.

The techniques for generating grammatically correct text are a bit beyond our immediate means (though not so very much so), so we're going to concentrate on using a bit of randomness to try and generate individual words which look plausible. In the end we'd like to see output like "anterite" but not like "xputzeiw".

Problem description

The basic problem is "generate a random word of a given length". To facilitate this we've provided an interface called **`WordGenerator`** which specifies exactly that that is what is required.

We will be generating words in lower case using only the characters **`'a'`** through **`'z'`**. This allows a little trick (or if you prefer "hack") which relies on the fact that internally to Java **`char`** is really an integer type. So for instance **`(char) ('a' + 4)`** is **`'e'`**. In the **`BasicGenerator`** class (also supplied) we simply choose letters uniformly at random in the required range by generating, for each letter, an **`int`** between 0 and 25 (inclusive), adding it to **`'a'`** and treating the result as a **`char`**.

The problem with **`BasicGenerator`** is that a typical run of the words it outputs might look something like: icdwpys, onjaxdm, xqlvaho, bfrhqfc, gftyjcg. This hardly meets our criterion of "looking plausible".

The first issue to address is that the letters in English words do not all occur with the same frequencies. So, the next thing to try is to generate words where the letter frequencies match those from English text (this will be "Part one"). To do this, you need to know how to choose an item randomly from a list, when each item has an associated weight (or probability) and the weights add up to 1. If we think of the items as belonging to an array, then each will have an index. We will also have an array, w, of weights, and we want to choose a random index according to the weights. The basic idea is to choose a random number between 0 and 1, and then choose the first index such that the sum of the weights up to that index is larger than the random number chosen. The following pseudocode describes how to do that:

chooseIndex(w):
  $i \leftarrow 0$
  $r \leftarrow$ a random number between 0 and 1
  while $r > w[i]$ **do**

$r \leftarrow r - w[i]$
$i \leftarrow i + 1$
end while
return i

Unfortunately a generator based on letter frequencies is not much of an improvement. A typical run might be: cirdesw, mhosstg, eietriw, otnlilt, nardmoi. At least one of these looks good (the last one) but the others have two letter combinations (or digrams) like 'mh', 'iw', 'nl' that are never, or at best very rarely, seen.

The idea of the next generator is to remedy this by choosing the first letter of each word according to the frequency with which letters occur as first letters, and from then on choosing a letter according to the frequency by which letters follow the preceding letter - that is, if we currently have letter say `'d'` we will look at the frequencies of all the pairs `'da'`, `'db'`, ..., `'dz'` and choose our next letter according to them.

You might be able to find a 26×26 table giving these frequencies online, but we'll take a different and more flexible approach. Start with a large chunk of English text. Process it one character at a time – for each pair of letters you see, add the second one to a `String` associated to the first one. Now, when you need a letter to follow `'d'` just take the `String` associated with `'d'` and choose a random character from it.

Here's a simple example - suppose that our "large chunk" of text was just the word "abracadabra". Then after processing we'd have the following:

Letter	String
a	bcdb
b	rr
c	a
d	a
r	aa

This wouldn't be very useful! Suppose we start with `'a'`. Half the time the next letter would be `'b'` (always followed by `'r'` and then by `'a'`). One quarter of the time (each) it would be `'c'` or `'d'` always followed by `'a'`. And then we'd begin again.

But of course the problem with this is that our "large chunk" was much too small and much too regular. In the second part of the lab we'll provide a set of strings harvested from a much larger chunk of text and leave the rest to you.

Provided files

We have provided you with a number of files to help implement your solutions to this week's tasks. The directory `/home/cshome/coursework/241/pickup/08` contains

a number of files which you should copy into your ~/241/08 directory before you begin. The files **WordGenerator.java** and **BasicGenerator.java** have already been described above. In addition to this you will find

- **RandomWords.java** - an application class which you can use to help test your code.
- **FrequencyGenerator.java** - a skeletion of the class you must implement to complete part one.
- **letter-frequencies.txt** - a file containing the frequency of each letter in normal English text. You should read the numbers from this file into the array w described in the pseudo-code above when implementing your frequency generator.
- **DigramGenerator.java** - a skeletion of the class you must implement to complete part two.
- **first-letters.txt** - a file containing one thousand letters in the frequencies that they occur at the start of English words. Your digram generator should select the first letter of a word at random from this list (using the Random instance).
- **continuations.txt** - a file containing 26 lines (one for each letter of the alphabet). Each line contains a string of characters associated with the corresponding letter. So line 1 contains the characters which come after 'a', line 5 contains the characters which come after 'e', line 17 contains the characters which come after 'q' etc. When constructing a word your digram generator should select each subsequent letter by choosing a random letter (using the Random instance) from the list of characters associated with the previous letter.

Part one (1%)

Complete the class **FrequencyGenerator** to implement the basic frequency method described above. The letter frequencies[1] should be read into an array (or equivalent) from the file letter-frequencies.txt. The Random instance should be used to obtain the random numbers needed for the algorithm.

---

[1] Letter frequencies sourced from http://www.cryptograms.org/letter-frequencies.php

Part two (1%)

Complete the class **`DigramGenerator`** to implement the digram based method described above. The contents of **`first-letters.txt`** should be read into a string, and the contents of **`continuations.txt`** into an array of strings (or equivalent) from where they can be accessed in the manner described above.

Reflection and extension

- **`DigramGenerator`** is not too bad, but unfortunately we sometimes get words with "weird" last letters (because some frequent digrams rarely end a word in English) not to mention unusual beginnings. How might one fix these problems?
- To generate text you can move from digrams of letters, to pairs (or triples) of words. How well does that work?
- How would you implement the "digram gathering" function? What effect does the initial chunk of text have?