The Problem

You have been asked to create a program that can take a text file as input and generate some text as output that has some similar characteristics as the input. It is not necessary that the output represent correct language or even be meaningful, but it needs to contain similar words in similar patterns. As a part of your program, you need to create at least two Java classes. One of these represents the model, which processes the input and provides text generation capability. The other represents the command-line application that allows the user to interface with the model. Specific requirements are detailed below.

To simplify the issues related to dealing with capitalization and punctuation (periods at the end of sentences, apostrophes for possessives/contractions, etc), we'll tweak the input when building our model to reduce the number of different words we consider. For the purposes of this assignment, words cannot contain whitespace or punctuation. If a sequence of characters without whitespace contains punctuation characters, the "word" is defined as the characters before the first punctuation character, if any.

As an example, in this string: "I'm a little Teapot short and stout. Here's my handle, 'ere's my spout" the words are:

[i, a, little, teapot, short, and, stout, here, my, handle, my, spout]

All letters are lowercased, and anything after a punctuation character is dropped. Note that "'ere's" contributes no words to the output because there are no letters before the apostrophe.

This "word simplification" strategy should help us produce more interesting outputs from smaller training input files.

For this assignment, we are defining the probability of a word coming after another as the number of times this occurred in the input text divided by the total number of words that came after the first word. This is a number between 0.0 and 1.0 inclusive. This means that somewhere in your code you need to be keeping track of this probability for each possible two-word sequence. There are many ways to accomplish that. Consider the most efficient data structures and algorithms for this application while considering the required functionality.

Requirements

- Create a new class named *TextGenerator* in a new package called *comprehensive*. The user starts your program by running this class (i.e., by executing *TextGenerator*'s *main* method). There are several command-line arguments required that must match these specifications exactly and in this order.

    - **First argument -** The path and name of the input text file. This can be an absolute or relative path.
    - **Second argument -** The "seed" word. We'll use your model to predict which word(s) come after the seed. You can assume that the seed is a word contained in the input file
    - **Third argument** - The number of words your program should generate as output (we'll call it K). This must be an integer that is not negative and would not overflow the int type.
    - **(Optional) Fourth argument -** The fourth argument determines what your program should output
        - **If there is no fourth argument**, the output will be the K most probable words that could come after the seed word. The words must be output in descending order from most probable to least. If there are fewer than K possible next words for the first word, fewer than K words will be output. For an input file containing "hello world" example, with seed "hello", if K = 3 the output would just be one word: "world".
        - If there is a fourth argument present, it must be either the word `all` or the word `one`. If it is `all`, the output will be generated while selecting randomly from all possible next words based on their probabilities. If it is `one`, generation will only select the next word with greatest probability. Ties for greatest probability will be broken using the lexicographical ordering of the words. For example, if "apple" and "zoo" were tied for greatest probability, the word "apple" would be selected. This means that the output using argument `one` is deterministic, while the output for `all` is not. In either case, the first word in the output must be the "seed" word. If you encounter a situation where there is no possible next word and you need to generate more words, the next word will again be the seed word. For example, If the input contains the text "hello world", and the number of needed output words is 7, and the seed is "hello", then the output will be "hello world hello world hello world hello".