CPS 109 Assignment Two, Fall 2017: Snowman

Learning objectives

The learning objectives of this assignment are, as in Big Java:

1. To operate on arrays of strings and characters.
2. To implement and make decisions based on string pattern matching.
3. To write methods that maintain data and state between the method calls in class variables.
4. To reason about and solve a real and interesting open-ended problem from computational linguistics that is easy to start out using simple code and logic, but allows virtually infinite room for further improvements and optimizations.

The game of Snowman

In this assignment, you implement a player for a familiar children's game in which your program tries to guess a secret word one letter at a time. For example, if the secret word is "snowman", then the player begins with the letters hidden: "*******". Each guess that does not appear in the word is a miss and scores one point. Each guessed letter which does appear in the word replaces the corresponding asterisks with the correct letter. For example, a guesss of 'n', results in "*n****n". The fewer misses your program does overall, the better. The guessing ends when all of the letters have been guessed, to reveal the hidden word, in this case "snowman".

To be able to write and execute this assignment, you need to download the following files and save them to the same directory in which you write your project:

- words.txt, the 2.5 megabyte raw text file containing the list of possible words in the English language, one word per line. From this dictionary, only the proper nouns that contain only lowercase letters are used in the game.
- SnowmanRunner.java, the automated tester for your Snowman code.

Class specification and methods to write

You must write and submit on D2L exactly one Java sourcecode file named SnowmanPlayer.java. This class **must** have all four of the following methods, **with names and signatures exactly as specified below**. (Otherwise, our SnowmanRunner tester class can't be compiled and executed with your class.) In addition, you may write in your class whatever private utility methods and data fields you wish. Furthermore, you are allowed to use and call any existing classes and methods in the Java standard library, such as ArrayList or various String processing methods.

Your methods must operate in silence and not print anything on the console, so as not to mess up the output of the tester class. Of course you can do some debugging prints during the development phase, but before submitting your SnowmanPlayer, you must absolutely comment out all output statements in it.

public static String getAuthor()

This method must return the real name of the student in the form "Lastname, Firstname", same way as it appears in RAMMS. This simply helps the graders to keep track of who is who when they are running the programs, and not get all these classes with the same name mixed up.

public static void startGame(String[] words, int minLength, int maxLength, String allowedChars)

This method is guaranteed to be called exactly once by SnowmanRunner at the beginning of the program execution, and after that, never again. Inside this method, you can do whatever initialization computations you wish to perform to optimize your success in the game, such as counting the individual letter frequencies in words. The parameters are as follows:

- words is an array of Strings that contains all possible words of English that the secret words are randomly chosen from.
- minLength and maxLength are the minimum and maximum lengths of the secret words, respectively. Your code can safely assume that the secret word length is between these two, inclusive.
- allowedChars is a String that contains all possible characters that can occur in the secret words. (This would be the 26 lowercase letters of the English alphabet.)

public static void startNewWord(int length)

The method **startNewWord**(int) is called by SnowmanRunner at the start of each new round (which will have a new secret word). Inside this method, you can perform whatever initialization you need for guessing each individual secret word. The parameter length tells you the length of the secret word (so that you can ignore all the potential secret words that have a different length).

public static char guessLetter(String pattern, String previousGuesses)

The method **guessLetter**(String, String) is called by SnowmanRunner to ask your code what letter it would like to guess next for the current secret word. Your method answers by returning a char. The parameters of guessLetter() are as follows:

- pattern is a String that contains the current secret word so that each unknown letter is displayed as the character SnowmanRunner.BLANK (currently the asterisk character), and each known letter is displayed as that letter. For example, if the secret word is seven characters long, and you have previously made the successful guesses 'n' and 's', the pattern might be "sn****n".
- previousGuesses is a String that contains the characters that you have already guessed for this particular secret word, including both hits and misses. (Of course, it would be rather silly for your method to return as your guess some character from previousGuesses, or some character that is not in allowedChars.)

Using the SnowmanRunner class

The `main` method of the `SnowmanRunner` class tries out your `SnowmanPlayer` with a total of `SnowmanRunner.ROUNDS` secret words, and counts the total number of misses that your player made while guessing all these secret words correctly. This game is like golf in that the lower your number, the better you did in the game.

The random number generator of `SnowmanRunner` uses the fixed seed `SnowmanRunner.SEED` that ultimately determines the random words that are used as secret words. Having the runner produce the same set of words each time helps you test, debug and optimize your code. Of course, you should try out different seed values to give your `SnowmanPlayer` different secret words.

When the class variable `VERBOSE` is turned on, the `SnowmanRunner` prints out each secret word, followed by the letters guessed by the `SnowmanPlayer`. The misses are marked with an exclamation mark following that letter. After the secret word has been successfully guessed, the number of misses for that word is printed inside parentheses. If you want to measure your code over a statistically significant sample size (such as `ROUNDS = 1000`), just assign `VERBOSE = false` to see only the final result, instead of getting your console flooded with output.

An example run of `SnowmanRunner`, using Dr. Kokkarinen's private solution for `SnowmanPlayer` with `SEED = 987654321` and `ROUNDS = 20`, produces the output like the following:

```
Read in 235886 words, of which 198529 remain after filtering.
Using seed 987654321, with 20 rounds.
hairmeal: es!ad!lc!himr (3)
pharmacomaniac: ioachmnpr (0)
regeneratrix: io!aegnrtx (1)
earwitness: etrainsw (0)
baccate: eatl!cb (1)
oinomania: e!ic!s!amno (3)
precompensate: i!eapcmnorst (1)
enlivenment: enilmtv (0)
azole: aei!olm!n!z (3)
ditchbank: e!io!atdbchkn (2)
anorganism: e!iorsagmn (1)
homonym: e!a!i!onmhy (3)
bruang: e!ant!gubr (2)
hexandry: ei!ao!yrndhx (2)
annulment: es!namtlu (1)
irrigate: eatin!gr (1)
ingrainedly: eiadglnry (0)
unlaunched: er!dunachl (1)
prevue: era!t!c!d!g!puv (5)
chariness: esniu!o!al!crh (3)
```

```
Code by 'Kokkarinen, Ilkka' made a total of 33 misses over 20 words.
```

When you start out this assignment with some simple logic, your total number of misses might be significantly larger than this; for example, the author's first quick and dirty working solution made a total of 186 misses (that is, an average of whopping 9.3 misses per word) over the same set of words! Simple optimizations to the logic of which letters you try next can rapidly cut down this number, but as your logic gets better and more accurate in guessing, the more difficult and smaller the further optimizations become. How far down does the optimum lie... who knows?

(As an aside, if there is anything that this exercise accidentally taught me, it is that I *really* don't know what most of the words in the English language even mean! Of the above twenty words, how many can you define, and when was the last time that you used each word in your speaking or writing? Most distributions that stem from real world emergent phenomena are highly spiky, with nothing uniform in them. *That is just how reality works*, and there is nothing that we can do about that other than adapt.)

Grading

When our grader TA marks your submission, we will use some different private seed that we will not be revealing in advance. (All student submissions are tested with the exact same secret seed and therefore the same secret words to guess, to keep this fair.) It is therefore pointless for you to hardcode the answers for some particular seed value into your methods. Later, this private seed will be revealed to all so that you can run the same secret words at home to verify that your code has been graded fairly and accurately. (Pseudorandom numbers have several nice applications in establishing trustworthy communication, as you will learn in your coming years of study.)

It is guaranteed that every possible word in the `words` array has the same probability of being chosen, and that the word does not change in the middle of guessing. You can therefore (pre-)compute whatever statistical properties of the words and character combinations of the English language to optimize your guessing.

The student submissions will be ranked in the ascending order of their total scores, and your assignment grade will be calculated on your placement in this overall ranking, according to a fair scale that will be decided later. Furthermore, as an added incentive, Dr. Kokkarinen will happily write a letter of recommendation for the top three students to whatever place they wish in the future, describing their achievement in this assignment.

Bonus marks: students whose submission produces a total score less than a threshold value (to be determined later) will receive bonus marks, provided their code has proper indentation and is well documented with **javadoc** comments (described on pages 36 and 207 of Big Java, Late Objects). The bonus can be up to 25%, meaning that one's score on a2 could be as high as 125%.

Plagiarism detection

You are to work alone when writing your code. You can discuss general ideas with your classmates, but you cannot copy code or develop code together nor take code from the web. We will be using the

Measure of Software Similarity (MOSS) to identify cases of possible plagiarism; see the following link for details: http://theory.stanford.edu/~aiken/moss. Note, MOSS can detect changing identifiers and rearranging code. The Department of Computer Science takes the act of plagiarism very seriously. Those caught plagiarizing (both originators and copiers) will be sanctioned. Please see Ryerson University's Policy 60 for possible penalties and consequences: http://ryerson.ca/senate/policies/pol60_procedures.pdf. If you are unsure what constitutes plagiarism, please see your instructor.