Homework 10 : Hangman

For HW10, you may work as a group (no more than 2 students).
Please mention your collaborator's name at the top of your code files.

This is a *2-part* homework assignment.

Part 1: Traditional Hangman

Traditional Hangman

Hangman is a 2-player word-guessing game generally played by having one player think of a word and the other player trying to guess that word letter by letter. Read more information about the game here.

Similar to Battleship, we want you to program a computer versus user version of the game. The computer will pick a random word from a dictionary that we will provide to you in a file. Note that we want you to use this dictionary file and only this dictionary file. It makes it easier for our testing.

- Represent the word to guess by using a row of underscores, representing each letter of the word (e.g., 'dog' would be shown to the user as _ _ _).
- The user guesses one letter at a time. Every correct letter that they guess should be shown in its correct location(s). Every guess that is incorrect will be shown in a list right underneath the word.
- Show the total number of guesses made. Allow the user to guess the same letter more than once. In such cases, give the user a friendly message and do not penalize them.

For example, below is an example of a reasonable interface. Feel free to make your program do exactly this or make it look even fancier!

```
Welcome to Hangman!

Guess a letter

_____
Total guesses: 0
a

Guess a letter

_____
Incorrect guesses: [a]
Total guesses: 1
c

Guess a letter

_____
Incorrect guesses: [a, c]
Total guesses: 2
e

Guess a letter
____e
Incorrect guesses: [a, c]
Total guesses: 3
i

Guess a letter
____e
Incorrect guesses: [a, c, i]
Total guesses: 4
e
You already guessed that letter!

Guess a letter
____e
Incorrect guesses: [a, c, i]
Total guesses: 4
```

Your Task

1. Read and parse the provided dictionary .txt file in Java, cleaning it up, if/when needed.

The word list file (*words.txt*) that we have provided to you in the canvas folder contains a number of words that do not really work for hangman. As part of your program we want you to read that file in and ensure that the computer never chooses a word that contains any of the following:

a) Upper case letters (Zambia, Shakespearean, Berlin, Mars, Navajo, deWeert)

b) Abbreviations - designated by a '.' (mrs., govt., dr., lb., adj.)

c) An apostrophe (you're, couldn't, won't, it's)

d) A hyphen (user-generated, custom-built, mother-in-law, editor-in-chief)

e) Compound words - words with spaces (post office, real estate, attorney general)

f) A digit (2nd, 3D, 12-sided)

NOTE: For your convenience, we are also providing a file (*words_clean.txt*), which only has "approved" words. It is a very short file, compared to the full *words.txt*. This should allow you to work on and quickly test out other parts of the assignment even if you haven't cleaned up the full dictionary.

2. Design the traditional hangman program

We are not going to provide you with a specific design for this homework, with a few exceptions regarding the required package structure. You must have a "dictionary" package, a "hangman" package, and of course, the default package.

a) You need a class that reads and parses the dictionary file and removes unacceptable words. Put this class in the "dictionary" package.
b) You need a class (or collections of classes) that defines the functionality of traditional hangman. Put this class (or classes) in the "hangman" package.
c) There should be a `HangmanGame` "controller" class that launches the game, and continuously takes user input until the game is over. Put this class in the default package.

Last, but not least, if you are not given a design to stick to, it might be in your best interest to start with a piece of paper and lay out what your classes and methods will be. In other words, do not dive into Eclipse and expect things to work out without some forethought.

Part 2: Evil Hangman

Evil Hangman

The evil version of this game basically exploits the computer's ability to store a large amount of information. In the traditional version of the game, the computer has to stick to the original word as the user guesses. In the evil version, the computer keeps changing the word in order to make the user's task harder.

The algorithm that drives Evil Hangman is fairly straightforward. The computer begins by picking a random word from the entire clean list of words, and maintaining a list of only the words of that length. Whenever the player guesses, the computer partitions the words into "word families" based on the positions of the guessed letters in the words of a particular length. For example, if the full word list is HEAL, HELP, BELT, HAIR, and MAKE and the player guesses the letter 'E', then there would be four word families:

- E E -, containing H**EE**L
- E - -, containing H**E**LP and B**E**LT
- - - -, containing HAIR
- - - E, containing MAK**E**

Once the words are divided into these groups, the computer picks the largest of the groups (with the most words) to use as its remaining word list. By doing so, it gives itself the maximum chance to dodge the user's guesses.

It then reveals the letters in the positions indicated by the word family. In this case, the computer would pick the family - E - - (because it has 2 words and the other families containing an E only have 1 word each). The computer would keep only the - E - - family of words and reveal an E in the second position of the word to the user.

If at any point there is a tie between families with the same number of words, the computer can just select one of the word families, either randomly or by selecting the first family.

The computer **cannot** reverse its decision once it has determined which letters to reveal. In the example above, if the user guesses 'L' next, the computer must pick a word from its word list that has four letters and contains an E in the second position. The computer cannot switch and pick the word HEEL. Since both HELP and BELT have an L, the computer must pick one of these two words and reveal the L in the correct position.

Your Task

Add the evil hangman functionality (as described above) to your traditional hangman program. When you launch the game from the `HangmanGame` class, the computer should decide which version of the game to play with the user. Your goal as the programmer is to ensure that, for the entire game, the user never knows what version of hangman is being played (evil or traditional). Do not print anything that might let the user discover this. Everything should look exactly the same to the user when playing evil hangman as it does when playing traditional hangman. At the end of the game, if/when the user wins, you can tell them which version of the game they were playing.

From a programming perspective you want evil hangman and traditional hangman **to share as much code as possible**. We want you to think hard about how you can accomplish this. The answer lies within the scope of the object-oriented concepts we have covered in this course.

For example, **you might** have an abstract class `Hangman`, with a subclass `HangmanTraditional` for the traditional version of the game and a subclass `HangmanEvil` for the evil version of the game. Put all of the classes that define the functionality for both traditional hangman and evil hangman in the "hangman" package.

Testing

Regardless of your design, we expect you to write unit tests for every public method. The only public methods that you can leave untested are those that perform file I/O, and simple getters and setters. You should keep the file I/O in as few methods as possible.

In order to perform tests, please use the smaller list of words (*words_clean.txt*). Reading in the larger text file (*words.txt*) in order to run unit tests is going to be inefficient.

Feel free to create additional versions of the dictionary .txt file for testing, but please be sure to include them with your homework submission.

Javadocs

Add Javadocs for all methods and variables. Create *API (Application Programming Interface)* documentation for your entire program. This can be extremely helpful for other programmers reading/running your code.

What to Submit

Please submit all the classes in your entire Java project. Make sure it includes everything in your "src" folder, both traditional and evil hangman, and a separate folder with all of your generated Javadoc HTML files.

Also include the members of your team as part of the @author tag in the Javadocs for your `HangmanGame` "controller" class.

Evaluation

You will be graded out of 40 points :

- Code writing, functionality, and design (*20 pts*)
  - Did you set up your program with the basic required package structure?
  - Does traditional hangman work as expected?
  - Does evil hangman work as expected?
  - Does the computer choose which version of the game to play and can the user tell?
  - Did you make good design decisions about code re-use?
    - How much code is being shared between traditional and evil hangman?
    - How is this code sharing being achieved in your design?
  - Does the code take too long to run? (10 seconds would be too long!)
- Loading, parsing, and navigating dictionary (*10 pts*)
  - Did you keep file I/O in as few methods as possible?
  - Did you accurately clean the dictionary file based on the provided guidelines?
  - What data structure(s) did you use?
- Unit testing (*5 pts*)
  - Did you write unit tests for every public method (excluding methods that perform file I/O and simple getters and setters)?
- Style & Javadocs (*5 pts*)
  - Adding Javadocs to all methods and variables, and comments to all non-trivial code
  - Generating HTML files from the Javadocs in your code using Eclipse