COSC-211: DATA STRUCTURES
HW6: SCRABBLE HELPER

Due Thursday, April 11, 11:59pm

Reminder regarding intellectual responsibility: This is an individual assignment, and the work you submit should be your own. Do not look at anyone else's code, and do not show anyone your code (except for me and the course TAs). You can (and should) look up the Java API documentation for any classes you're considering using in this assignment. Do NOT look on the internet (or anywhere else) for code that solves the problem in this assignment.

1 The Assignment

In the game Scrabble, players take turns placing tiles on a board to form words. Each player has a set of seven letters in front of them, and must use these letters to form a word attached to the words that already appear on the board. A key part of the game, thus, is finding *anagrams*: given a set of letters, we want to find all possible words that can be made using those letters. For example, if our letters are `aemt`, we could make any word in the set `mate, meat, meta, tame, team`. (In the real game of Scrabble we'd also want to find shorter words like `mat` and `eat`, but for the purposes of this assignment we'll stick to words that use *all* of the specified letters.)

Your job in this assignment is to write a program to help a Scrabble player find anagrams. The idea is that you'll read a list of words from a text file, and group these words according to the set of letters they use (so the set of words `mate, meat, meta, tame, team` should all be in the same "group," associated with the set of letters `aemt`).

Some specifications for how your program should run:

- Your program must be in a file called `ScrabbleHelper.java`.
- Your program must take as a command line argument the text file in which your list of words is stored. That is, I should be able to run your program with the command:

  ```
  java ScrabbleHelper myDictionary.txt
  ```

  where the list of valid words is stored in the file `myDictionary.txt`.
- Your program should repeatedly prompt the user to enter a word, then print out all valid words that are anagrams of the entered word, then ask the user if they want to test another word. This process should continue until the user types "no". For example:

  ```
  Enter your letters:
  team
  mate meat meta tame team
  Another?
  ```

```
yes
Enter your letters:
life
feil fiel file lief life
Another?
yes
Enter your letters:
ear
aer are ear era rea
Another?
no
```

Notice that when the user types in `life`, the program does not print out words like `eilf` or `flie`, because these are not valid words (i.e., they do not appear in the dictionary).

Some specifications for what your program should do:

- Read in all of the words from the specified text file. On Linux servers (e.g., `remus/romulus`) there's a dictionary stored in the file `/usr/share/dict/words`. Macs and Windows also have built-in dictionaries (feel free to Google to look up how to find them), or you can use your own list of words (this might be helpful, in particular, for debugging).
- Store all of the valid words (i.e., the words that appear in your dictionary) in a `HashMap` (this is a Java library; do not write your own `HashMap` class!). The `key` should be a `String` containing the letters in a word sorted in alphabetical order, and the `value` should be a `Vector` (or your preferred form of list) of all valid anagrams of that `key`. For example, the `<key,value>` pair for our example above is `<aemt, mate, meat, meta, tame, team>`.
- Once you have this `HashMap` of anagrams, you can start prompting the user to enter words, as described above.

In contrast to many of our earlier assignments, your goal here is to practice using the data structures and classes contained in existing Java libraries, rather than to build your own versions of these data structures from scratch. You should look at the online documentation for any classes that you're considering using. In particular, you might find the HashMap, Vector, and String documentation particularly helpful. But feel free to look up other classes!

1.1 Code Style and Formatting

Please pay attention to the style of your code as well as its functionality. Think of your code as you would an essay: someone else is going to read it, and therefore you want it to be clear and easy to read. Some things to consider:

- Comment your code! Note what each method and large block of code is supposed to do. You don't have to (and probably shouldn't) comment every single line, but it's very helpful

to use comments to give a "road map" to help someone reading your code understand what you intend it to do.

- Use consistent indentation to indicate blocks of code. Your code should look like this:

```
public void myMethod() {
    for(int i = 0; i < 10; i++) {
        for(int j = 0; j < 10; j++) {
            System.out.println(i + j);
        }
    }
}
```

and not:

```
    public void myMethod() {
        for(int i = 0; i < 10; i++) {
        for(int j = 0; j < 10; j++) {
    System.out.println(i + j);
    }
        }
     }
```

Which do you think is easier to understand?

- Use helpful variable and method names. If a variable is meant to count the number of words you've read in from a file, call it `wordCount` and not `wxyz`.

2 Submit your work

Make sure you **test your code thoroughly** before submitting it. Code that does not compile will not receive credit.

Submit all of your Java files using either the submission web site or (from `remus/romulus`) the command line:

```
$ cssubmit *.java
```

This assignment is due on Thursday, April 11, 11:59pm.