CSE 143 Assignment 4 (Hangman) Score Sheet

Student(s): akshit <akshit@uw.edu>

Graded by: Melissa Medsker <medskm@cs.washington.edu>

28 / 30 : Total Score

28 / 30 : Correctness

3 / 3 : fields and getters

1 / 1 : words

1 / 1 : guessesLeft

1 / 1 : guesses

4 / 4 : constructor

2 / 2 : word set is initialized properly

1 / 1 : pattern correct; has exactly length number of dashes

1 / 1 : guesses and guessesLeft correct

9 / 9 : record

4 / 4 : words works

1 / 1 : words works in basic cases

2 / 2 : words works in the general case without ties

1 / 1 : words works with ties

1 / 1 : guesses is correct for 2 out of 3 test cases

1 / 1 : pattern works in basic cases for 2 out of 3 test cases

1 / 1 : guessesLeft works in basic cases for 2 out of 3 test cases

2 / 2 : everything but the set works in the general case

Nice work!

3 / 3 : exceptions

3 / 3 : method decomposition

1 / 1 : attempt

2 / 2 : good decomposition

Good work factoring out methods for clean method decomposition!

2 / 3 : comments

1 / 1 : attempt to comment

1 / 2 : well documented code

-1: See yellow

Excellent use of inline comments!

4 / 5 : otherwise good style

-1: See green

Lateness and Other Deductions

	Thu 2016/10/27 11:30pm	Due
	Thu 2016/10/27 09:09pm	Submitted (on time)
0	Late days used on this assignment	
0	Lateness deduction	
	Other deductions	

Overall comments:

Great work Akshit!

Annotations: HangmanManager.java

```
        10        20        30        40        50        60        70        80
/**
 * @author Akshit Patel
 * @Date 10/21/2016
 * CSE 143D DC
 * TA: Melissa Medsker
 * HW #4 Evil Hangman
 */
import java.util.*;//sets & maps.

/**
 * This class manages an evil game of Hangman. Evil as it initially does not
 * choose one single word instead a set of words and eventually gets to one
 * after every guess depending on the character guessed by the player, the class
 * also keeps track of the characters guessed, the number of guesses left and
 * provides with the pattern of the word choice with letter guesses in right
 * occurrence order when guessed correctly.
 *
 */
public class HangmanManager {

    private int guesses;// keeps check of the guesses left.
    private Set<String> wordGuesses;// words used in game.
    private Set<Character> guessLetters;// letters already guessed.
    private String pattern;// stores the pattern of the answer.

    /**
     * This constructor method initializes the Hangman game by selecting the
     * words of specific length also creating a guess pattern full of dashes
     * till the word length indicating no correct guess made, and also
     * initializes the number of guesses allowed for one game.
     *
     * @param dictionary list of string of words that can be used for one game
     * of Hangman.
     * @param length int value of the size of the secret word.
     * @param max total number of guesses allowed for this game.
     * @throws IllegalArgumentException if the length of words is less than 1 or
     * if the max guesses are less than 0;
     */
    public HangmanManager(List<String> dictionary, int length, int max) {
        if (length < 1 || max < 0) {
            throw new IllegalArgumentException();
        }
        this.guesses = max;// total guesses allowed.
        this.guessLetters = new TreeSet<Character>();// Initialize guesses made.
        this.wordGuesses = new TreeSet<String>();// Initialize words used.
        // get words of specific length from the List provided.
        for (String word : dictionary) {
            if (word.length() == length) {
                this.wordGuesses.add(word);
            }
        }
        this.pattern = "";// Initialize empty pattern.
        while (this.pattern.length() != length) {
            this.pattern += "-";// add dashes initially.
        }
    }

    /**
     * This method helps to get the words being managed for the game.
     *
     * @return set of words used in the game.
     */
    public Set<String> words() {
        Set<String> copyWords = new TreeSet<String>();
        for (String word : this.wordGuesses) {
            copyWords.add(word);
        }
        return copyWords;
    }

    /**
     * this method helps to get the total number of guesses left for the game
     * played.
     *
     * @return int number of guesses remaining.
     */
    public int guessesLeft() {
        return this.guesses;
    }

    /**
     * This method gives the characters already guessed for the game played.
     *
     * @return set of letters guessed in this game by the player.
     */
    public Set<Character> guesses() {
        Set<Character> copyGuesses = new TreeSet<Character>();
        for (char letter : this.guessLetters) {
            copyGuesses.add(letter);
        }
        return copyGuesses;
    }

```

-0: This comment should be more clear that the method *does* return this Set

-0: Rather than writing a for-each loop to add each individual element in these Set fields, it would be more concise to use the Set constructor which takes another Set as a parameter

```java
    /**
     * This method gives correct pattern of guesses made, full of dashes if no
     * correct guesses or mix of dashes with correct guessed character in order
     * of occurrence.
     *
     * @return representation of the pattern of correct or incorrect guesses.
     * @throws IllegalStateException if the no words are managed i.e. the set of
     * words managed is empty.
     */
    public String pattern() {
        if (this.wordGuesses.isEmpty()) {
            throw new IllegalStateException();
        }
        return this.pattern;
    }

    /**
     * This method manages the Hangman game by keeping track of the guess made
     * by the player, updating the guesses when a wrong choice of letter is made
     * and accordingly chooses words to with more choices and gives info about
     * the number of occurrences of the guessed letter in the new pattern of the
     * guess word(s) considered.
     *
     * Pre-Condition: The guesses made are lowercase alphabetic letters.
     * Post-Condition: The set of words is updated to the one with more choices.
     *
     * @param guess the character guessed by the player in the game.
     * @return number of occurrences of the correct guess in the secret word.
     * @throws IllegalStateException if guesses left are less than 1 or if the
     * set words used for the game is empty.
     * @throws IllegalArgumentException if the character is already guessed by
     * the player & the set of words is not empty.
     */
    public int record(char guess) {
        if (this.guesses < 1 || this.wordGuesses.isEmpty()) {
            throw new IllegalStateException();
        }
        if (!this.wordGuesses.isEmpty() && this.guessLetters.contains(guess)) {
            throw new IllegalArgumentException();
        }
        this.guessLetters.add(guess);// add the guess.
        // Map to set pattern as keys and the words as associated values.
        Map<String, Set<String>> wordMap = new TreeMap<String, Set<String>>();
        // generate the guess words and the pattern.
        this.generateGuesses(guess, wordMap);
        // get set of words with most choices.
        this.getWords(wordMap);
        // return occurrence and update guesses left if 0 occurrence.
        return this.getOccurence(guess);
    }

    /**
     * This method creates the possible patterns for every set of words with the
     * every choice available.
     *
     * @param guess the character guessed by the player in the game.
     * @param wordMap map of pattern with associated words as values.
     */
    private void generateGuesses(char guess, Map<String, Set<String>> wordMap) {
        // make the pattern for every word and update the map.
        for (String word : this.wordGuesses) {
            String patternKey = "";// initialize the pattern for the word.
            for (int i = 0; i < word.length(); i++) {
                if (word.charAt(i) == guess) {
                    patternKey += guess;// add the letter guessed,
                } else {
                    // if guess letter is different then get the previous
                    // correct/incorrect guess like the correct letter or dash.
                    patternKey += this.pattern.charAt(i);
                }
            }
            // if pattern dosen't exist then add the pattern.
            if (!wordMap.containsKey(patternKey)) {
                // initialize the set of word for that pattern.
                Set<String> value = new TreeSet<String>();
                wordMap.put(patternKey, value);// add the pattern to map.
            }
            // add the current word to the set of words with similar pattern.
            wordMap.get(patternKey).add(word);
        }
    }

    /**
     * This method updates the set of words considered for the game to the one
     * with the most choices available
     *
     * @param cheatMap map of pattern with associated words as values.
     */
    private void getWords(Map<String, Set<String>> cheatMap) {
        // variable initialized to get the set of words with most choices.
        int size = 0;
        // get the set of words with more choices.
        for (String evilKey : cheatMap.keySet()) {
            int mapSize = cheatMap.get(evilKey).size();
            if (mapSize > size) {
                size = mapSize;// set the size of the larger set found.
                this.pattern = evilKey;// get the pattern of this.
            }
```

-0: This should be more clear in that only correctly-guessed characters are included in the pattern, and any characters that have not been guessed are represented as dashes

-0: This should be worded in a way that is more useful for a client to know (e.g., "narrows down possible word answers to maximize the computer's chance of winning based on the given guess")

-0: This check is unnecessary since wordGuesses will never be empty if your previous exception wasn't thrown

Great use of inline comments here!

-1: Missing comment on guess parameter

Since your given Map is always empty in this method, it would be better to handle its construction in this method, and return it after adding words to it

Great work with method decomposition here using helper methods!

-1: You should follow the add-to-map idiom introduced in class to avoid redundancy - here, you should first check if the family doesn't contain the key, where you would then add the new Set, and then outside of this if condition, add the word to wordMap.get(patternKey)

-0: What is the role of the Map in this method's behavior?

```
        }
        this.wordGuesses = cheatMap.get(this.pattern);// update the words.
    }

    /**
     * This method gives the number of occurrences of the letter guessed in the
     * answer pattern used for the game played. It also updates the guesses left
     * for this game when incorrect guess is made i.e. no occurrence of letter.
     *
     * @param guess the character guessed by the player.
     * @return int occurrences of the letter guessed in the correct answer
     * pattern.
     */
    private int getOccurence(char guess) {
        int occurences = 0;// Initialize the number of occurrence.
        // get the occurrence by going through the pattern selected.
        for (int i = 0; i < this.pattern.length(); i++) {
            if (this.pattern.charAt(i) == guess) {
                occurences++;
            }
        }
        // if no occurrences of letter then subtract the guesses left by one.
        if (occurences == 0) {
            this.guesses--;
        }
        return occurences;
    }
}

```