CSE 143: Computer Programming II — Autumn 2021

Take-home Assessment 4: Evil Hangman *due October 28, 2021 11:59pm*

many thanks to Keith Schwarz for this assignment

This assignment will assess your mastery of the following objectives:

- Implement a well-designed Java class to meet a given specification.
- Use sets (via the `Set<E>` interface) and maps (via the `Map<K, V>` interface) effectively.
- Choose an appropriate data structure to represent specified data.
- Create and manipulate nested collections.
- Follow prescribed conventions for code quality, documentation, and readability.

Overview: The Game of (Evil) Hangman

Sample execution log

```
Welcome to the cse143 hangman game.

What length word do you want to use? 4
How many wrong answers allowed? 7

guesses : 7
guessed : []
current : - - - -
Your guess? e
Sorry, there are no e's

guesses : 6
guessed : [e]
current : - - - -
Your guess? o
Yes, there are 2 o's

guesses : 6
guessed : [e, o]
current : - o o -
Your guess? d
Sorry, there are no d's

guesses : 5
guessed : [d, e, o]
current : - o o -
Your guess? c
Yes, there is one c

guesses : 5
guessed : [c, d, e, o]
current : c o o -
Your guess? l
Yes, there is one l

answer = cool
You beat me
```

In the game of hangman, one player (in our case, a computer) picks a word that another player (the user) is trying to guess. The guesser guesses individual letters until the word is fully discovered (in which case the guesser wins) or a specified number of incorrect letters is guessed (in which case the guesser loses). As correct letters are guessed, their location in the secret word is revealed to the guesser. You can learn more about the game of hangman on its Wikipedia page. (This game has been the inspiration for many other games, including the popular American gameshow *Wheel of Fortune*.)

In our *EVIL!* game of hangman, the computer delays picking a specific secret word until it is forced to do so. As a result, the computer is always considering a set of words that *could* be the secret word. In order to fool the user into thinking it is playing fairly, the computer only considers words with the same letter pattern.

For example, suppose that the computer knows the following words:

```
ally beta cool deal else flew good hope ibex
```

In our game, instead of beginning by choosing a word, the computer narrows down its set of possible answers *as* the user makes guesses.

When the user guesses 'e', the computer must reveal where the letter 'e' appears. Since it hasn't chosen a word yet, its options fall into *five* families:

Pattern	Family
- - - -	[ally, cool, good]
- e - -	[beta, deal]
- - e -	[flew, ibex]
- - - e	[hope]
e - - e	[else]

The guess forces the computer to choose a *family* of words, but not a particular *word* in that family.

The computer could use several different strategies for picking the family to display. Your program should always choose the family with the largest number of words. This strategy is reasonable because it leaves the computer's options open.

For the example described above, the computer would pick `- - - -`. This reduces the possible answers it can consider to:

```
ally cool good
```

Since the computer didn't reveal any letters, it counts this as a wrong guess and decreases the number of guesses left to 6. Next, the user guesses the letter 'o'. The computer has *two* word families to consider:

Pattern	Family
`- o o -`	`[cool, good]`
`- - - -`	`[ally]`

It picks the biggest family and reveals the letter 'o' in two places. This was a correct guess so the user still has 6 guesses left. The computer now has only two possible answers to choose from:

```
cool good
```

If the user guesses a letter that doesn't appear anywhere in your set of words, say 't', the family you previously chose is still going to match. In this case, you'd count 't' as a wrong answer.

When the user picks 'd', the possible families are all the same size (one word each). If there is a tie for largest family, the computer should choose the family whose pattern comes alphabetically first. In this example, the computer removes "good" from its consideration (because the pattern for to the family with "cool" is alphabetically before the pattern for the family with "good") and uses the family with "cool".

We have provided you with a client program, `HangmanMain.java`, that does the file processing and user interaction. It reads a dictionary text file as input and passes its entire contents to you as a list of strings. In this assessment, you will write a class `HangmanManager` that manages the state of a game of hangman.

HangmanManager

Your `HangmanManager` class should have the following constructor:

```
public HangmanManager(Collection<String> dictionary, int length, int max)
```

Your constructor is passed a dictionary of words, a target word length, and the maximum number of wrong guesses the player is allowed to make. It should use these values to initialize the state of the game. The set of words should initially contain all words from the dictionary of the given length, eliminating any duplicates. You should throw an `IllegalArgumentException` if the given length is less than 1 or if max is less than 0.

You may assume the given `Collection` contains only non-empty strings composed entirely of lowercase letters.

Your `HangmanManager` class should also implement the following methods:

```
public Set<String> words()
```

The client calls this method to get access to the current set of words being considered by the `HangmanManager`.

`public int` **`guessesLeft`**`()`
The client calls this method to find out how many guesses the player has left.

`public Set<Character>` **`guesses`**`()`
The client calls this method to find out the current set of letters that have been guessed by the player.

`public String` **`pattern`**`()`
The client calls this method to find out the current pattern to be displayed for the game, taking into account guesses that have been made. Letters that have not yet been guessed should be displayed as a dash and there should be spaces separating the letters. There should be no leading or trailing spaces. This operation should be "fast" in the sense that it should store the pattern rather then computing it each time the method is called. This method should throw an `IllegalStateException` if the set of words is empty.

Save the pattern to avoid recomputation

`public int` **`record`**`(char guess)`
The client calls this method to record that the player made a guess. Using this guess, your method should decide what set of words to use going forward. It should return the number of occurrences of the guessed letter in the new pattern and it should appropriately update the number of guesses left. This method should throw an `IllegalStateException` if the number of guesses left is less than 1 or if the set of words is empty. If the previous exception was not thrown, it should throw an `IllegalArgumentException` if the character being guessed was guessed previously. You may assume that all guesses passed to the record method are lowercase letters.

Implementation Guidelines

Your program should exactly reproduce the format and general behavior described on the first page of this specification. You should use the `TreeSet` and `TreeMap` implementations for all sets and maps you make.

You may use any of the standard methods from the `String` class, but be careful not to introduce unnecessary inefficiency. You may not use regular expressions for this problem (if you don't know what that is, that's okay!). You may also not use the `toCharArray` method because it creates an unnecessary extra data structure (an array).

`guessesLeft()`

In Hangman, the player has a certain number of wrong guesses that they are allowed to make– this number is *not* the same as the total number of guesses they make. Correct guesses don't count against the user's guesses left since this value is the number of incorrect guesses the user can make before the game ends. *We will define the game as being over once guesses left is 0.*

`guesses()`

Note that the set `guesses()` returns is a set of `Character` values. Recall that you must use `Object` types inside < >, and `Character` is the wrapper class for `char` values. You may generally manipulate the set as if it were a set of simple `char` values (e.g., calling `add` or `contains` with a simple `char` value).

`record()`

For each call to `record`, you should find all the possible word families and pick the one with the most words. Use a `Map` to associate family patterns with the set of words that have each pattern. If there is a tie (two of the word families are of equal size), you should pick the one that occurs earlier in the Map (i.e., the one whose key comes up first when you iterate over the key set). The set of words representing the biggest family then becomes the dictionary for the next round.

Note that `Strings` are sorted alphabetically according to ASCII value. `'-'` comes before all letters of the alphabet according to these values.

Keep in mind that the patterns come from the words themselves. On any given turn, there is a current set of words that all have the same pattern. When the user guesses a new letter, go through each of the words that you have in the current set and figure out what the correct new pattern would be for that particular word given the new guess. You are likely to get different patterns for different words.

Your task is to process each of the words in the current set, putting each into a set that corresponds to the new pattern for that particular word.

Different words go in different sets because they have different patterns. Once you have processed all of the words, you go through the different sets and find the one with the most words. That becomes the new set used by the HangmanManager.

Development Strategy and Hints

The `record` method is the most difficult, so we strongly suggest you write it last. Create a simple test client to verify behavior as you add it. For example, write code to produce a Hangman-style clue (with dashes for letters that have not been guessed) and verify that it works in your simple client.

Another good task to complete and test in isolation is building the map that associates word family patterns to words in that family.

`HangmanMain` has two constants that you will want to change:

- `DICTIONARY_FILE` represents the name of the file to read the dictionary from. By default, it reads from `dictionary.txt` which contains over 127,000 words from the official English Scrabble dictionary. When getting started, you may find it helpful to change it to `dictionary2.txt` which contains the 9 words used in the example on the first page.
- `SHOW_COUNT` is set to `false` by default. Set it to `true` to see the words the computer is still considering as you play.

Notice that the `pattern` and `record` methods throw an exception when the set of words is empty. The only way this can happen is if the client requests a word length for which there are no matches in the dictionary or if the dictionary is empty to begin with. For example, the dictionary might not have any words of length 25.

On the last page if this specification is a diagram showing the decisions made by the computer in the game outlined on the first page. This may help you visualize how the families of words are chosen.

Code Quality Guidelines

In addition to producing the behavior described above, your code should be well-written and meet all expectations described in the grading guidelines, Code Quality Guide, and Commenting Guide. For this assessment, pay particular attention to the following elements:

Avoid Redundancy

Factor out any redundancy in your methods.

Create "helper" method(s) to capture repeated code. As long as all extra methods you create are `private` (so outside code cannot call them), you can have additional methods in your class beyond those specified here. If you find that multiple methods in your class do similar things, you should create helper method(s) to capture the common code. In particular, **for this assessment you should not have any methods**

that have more than 20 lines of code in their body (not counting blank lines and lines that have just comments or curly braces). If you have a method that requires more than 20 lines of code, then you should break it up into smaller methods.

Generic Structures

You should always use generic structures. If you make a mistake in specifying type parameters, the Java compiler may warn you that you have "unchecked or unsafe operations" in your program. If you use jGRASP, you may want to change your settings to see which line the warning refers to. Go to `Settings/Compiler Settings/Workspace/Flags/Args` and then uncheck the box next to "`Compile`" and type in: `-Xlint:unchecked`

Data Fields

Properly encapsulate your objects by making data your fields `private`. Avoid unnecessary fields; use fields to store important data of your objects but not to store temporary values only used in one place. Fields should always be initialized inside a constructor or method, never at declaration.

Exceptions

The specified exceptions must be thrown correctly in the specified cases. Exceptions should be thrown as soon as possible, and no unnecessary work should be done when an exception is thrown. Exceptions should be documented in comments, including the type of exception thrown and under what conditions.

Commenting

Each method should have a header comment including all necessary information as described in the Commenting Guide. Comments should be written in your own words (i.e. not copied and pasted from this spec) and should not include implemenation details.

Running and Submitting

If you believe your behavior is correct, you can submit your work by clicking the "Mark" button in the Ed assessment. You will see the results of some automated tests along with tentative grades. **These grades are not final until you have received feedback from your TA.**

You may submit your work as often as you like until the deadline; we will always grade your most recent submission. Note the due date and time carefully—**work submitted after the due time will not be accepted**.

Getting Help

If you find you are struggling with this assessment, make use of all the course resources that are available to you, such as:

- Reviewing relevant examples from class
- Reading the textbook
- Visiting support hours
- Posting a question on the message board

Collaboration Policy

Remember that, while you are encouraged to use all resources at your disposal, including your classmates, **all work you submit must be entirely your own**. In particular, you should **NEVER** look at a solution to this assessment from another source (a classmate, a former student, an online repository, etc.). Please review the full policy in the syllabus for more details and ask the course staff if you are unclear on whether or not a resource is OK to use.

Reflection

In addition to your code, you must submit answers to short reflection questions. These questions will help you think about what you learned, what you struggled with, and how you can improve next time. The questions are given in the file `EvilHangmanReflection.txt` in the Ed assessment; type your responses directly into that file.

Appendix: Diagram of Computer's Choices