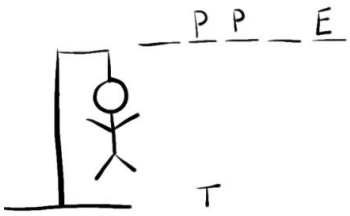
CS145 – PROGRAMMING ASSIGNMENT #3

EVIL HANGMAN



OVERVIEW

This program focuses on programming with Java Collections classes. You will implement a non-static version of the class Hangman word guessing game in in a file named HangmanManager.java. You will also practice exception throwing and string manipulation.

Your program will need support files HangmanMain.java, dictionary.txt, and other input files downloadable from the class canvas website.

INSTRUCTIONS

In a normal game of hangman, the computer picks a word that doesn't change, and then the user is supposed to guess. The user guesses individual letters until the word is fully discovered. If you aren't familiar with the general rules of the game of hangman, review its Wikipedia page: http://en.wikipedia.org/wiki/Hangman_(game).

In our game of hangman, the computer delays picking a word until it is forced to. As a result, the computer is always considering a set of words that could be the answer. 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 words in the following dictionary

ALLY BETA COOL DEAL ELSE FLEW GOOD HOPE IBEX

In a normal game of hangman, the computer would start the game by choosing a word to guess. In our game, the computer doesn't yet commit to an answer but instead narrows down its set of possible answers as the user makes guesses. In the log below, the user's first guess is 'e'. The computer has to reveal where the letter 'e' appears but since it hasn't chosen an answer, it has several options. In particular, note that the dictionary's words fall into 5 families:

- `----` is the pattern for `[ally, cool, good]`
- `e---` is an empty pattern with no possible values.
- `-e--` is the pattern for `[beta, deal]`
- `--e-` is the pattern for `[flew, ibex]`
- `---e` is the pattern for `[hope]`
- `e--e` is the pattern for `[else]`
- `all other patterns are empty.`

The computer could choose to reveal any of these 5 patterns. It could use several different strategies for picking the family to display. **For this project, your program will always choose the largest of the remaining word families**. This will leave the computer's options open and increase its chances of winning. 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:

- `-oo-` is the pattern for `[cool, good]`
- `----` is the pattern for `[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`

When the user picks 'c', the computer removes 'cool' from its consideration and is now locked into using 'good' as the answer. With a large dictionary, it will take much longer for the computer to have to pick an answer!

SAMPLE EXECUTION

```
Welcome to my 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? c

Sorry, there are no c's
guesses : 5
guessed : [c, e, o]
current : - o o -
Your guess? g
Yes, there is one g

guesses : 5
guessed : [c, e, g, o]
current : g o o -
Your guess? d
Yes, there is one d
answer = good
You beat me
```

GETTING STARTED

In order to get started with this assignment, start by downloading the CS145 Assignment 3 ZIP file from Canvas.

File	Actions
dictionary.txt	The primary list of words for the game to use.
dictionary2.txt	A small sample size dictionary that you can use to test your program.
HangmanMain.java	This is the file that you will run. No major modifications can be made to this file. However you can change the dictionary name and turn on debug mode.

YOUR INSTRUCTIONS

You are given a client program HangmanMain.java that does the file processing and user interaction. HangmanMain reads a dictionary text file as input and passes its entire contents to you as a list of strings. You are to write a class called HangmanManager that will manage the state of the game.

NECESSARY METHODS

PUBLIC HANGMANMANAGER(LIST<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 be all words from the dictionary that are of the given length. You should throw an `IllegalArgumentException` if length is less than 1 or if max is less than 0.

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 SORTEDSET<CHARACTER> GUESSES()

The client calls this method to find out the current set of letters that have been guessed by the user.

PUBLIC STRING PATTERN()

This method should return the current pattern to be displayed for the hangman 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 method should throw an `IllegalStateException` if the set of words is empty.

PUBLIC INT RECORD(CHAR GUESS)

This is the method that does most of the work by recording the next guess made by the user. Using this guess, it 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 not at least 1 or if the list of words is empty. It should throw an `IllegalArgumentException` if the list of words is nonempty and the character being guessed was guessed previously.

OTHER INSTRUCTIONS

Your program should exactly reproduce the format and general behavior demonstrated on the previous pages.

You may assume that the list of words passed to your constructor will be a nonempty list of nonempty strings composed entirely of lowercase letters. You may also assume that all guesses passed to your record method are lowercase letters.

Notice that the `guesses` method returns a `SortedSet` rather than a Set. `SortedSet` is a variation of the `Set` interface that requires that the values in the set are to appear in increasing order. We want this property for the set of guesses so that we can show the user the guesses in alphabetical order. The `TreeSet` class implements this interface, just as `TreeSet` implements the `Set` interface. Also notice that the set returned is a set of `Character` values. Recall that type parameters must be object types and that `Character` is the wrapper class for char values. You can 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).

The `pattern` and `record` methods throw an exception when the list 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. For example, the dictionary does not have any words of length 25.

For each call on `record`, you will find all of the possible word families and pick the one that has the most elements. 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.

Use the `TreeSet` and `TreeMap` implementations for all of your sets and maps.

HINTS

The hardest method is `record`, so write it last.

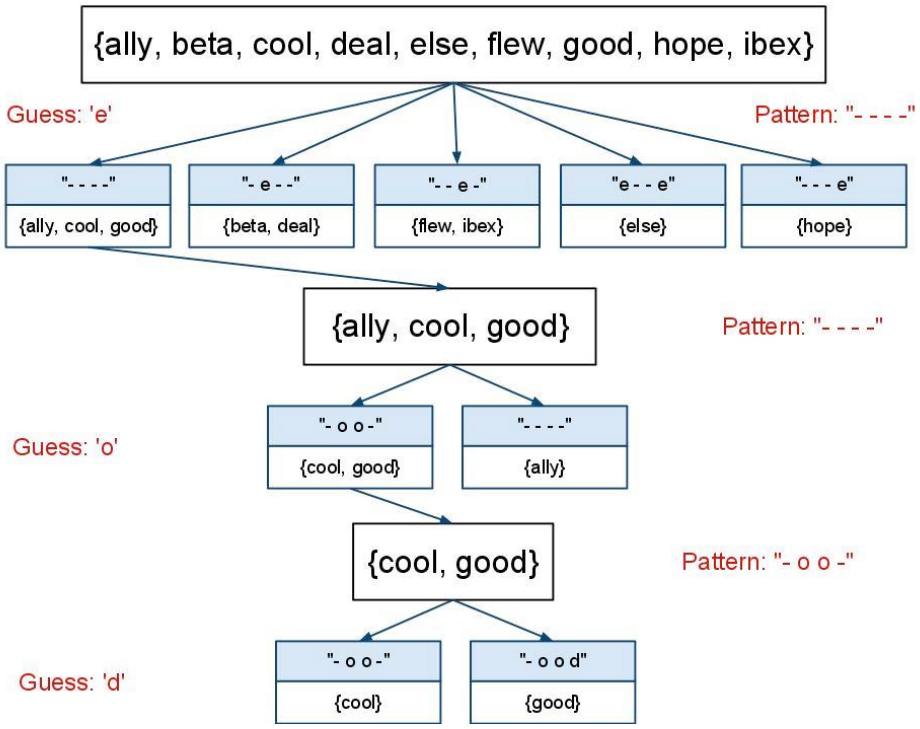
`HangmanMain` has two constants that you will want to change. The first represents the name of the dictionary file. By default, it reads from dictionary.txt which contains over 127,000 words from the official English Scrabble dictionary. When getting started, change it to dictionary2.txt which contains the 9 words used in the example on the first page. `DEBUG` is set to false by default. Set it to true to see the words the computer is still considering as you play.

One of the harder parts to this assignment is taking a word and then finding the "pattern" that goes with it. A good suggestion is to write a method that takes a string and a char and returns the associated pattern. For example `("Bookkeeper" , 'e')` would return `"-----ee-e-"`. You can then use this method in multiple places in your code.

If you have not had to deal with exceptions before the proper way to throw and exception is the following: `throw new IlleagalStateException().` This will end your method and return back to the primary program.

When you return the string from `Pattern()` you want to return a set of letters/dashes with spaces between them. However you will not want the spaces as part of the pattern while you work on it/use it. A suggestion would be to store the pattern without spaces, and only add the spaces in the `Pattern()` method before you return a modified version of it. (Store normally, add spaces only when asked for).

Here is a diagram showing the decisions made by the computer in the scenario described on the first page:


{ally, beta, cool, deal, else, flew, good, hope, ibex}
Guess: 'e'
Pattern: "- - - -"
"- - - -"
{ally, cool, good}
"- e - -"
{beta, deal}
"- - e -"
{flew, ibex}
"e - - e"
{else}
"- - - e"
{hope}
{ally, cool, good}
Pattern: "- - - -"
Guess: 'o'
"- o o -"
{cool, good}
"- - - -"
{ally}
{cool, good}
Pattern: "- o o -"
Guess: 'd'
"- o o -"
{cool}
"- o o d"
{good}

STYLE GUIDELINES AND GRADING:

You should introduce private methods to avoid redundancy and to break up large methods into smaller methods. In particular, **you should not have any methods that have more than 30 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 30 lines of code, then you should break it up into smaller methods.

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.

You should follow general good style guidelines such as: making fields private and avoiding unnecessary fields; declaring collection variables using interface types; appropriately using control structures like loops and if/else; properly using indentation, good variable names and types; and not having any lines of code longer than 100 characters. Comment your code descriptively in your own words at the top of your class, each method, and on complex sections of your code. Comments should explain each method's behavior, parameters, return, pre/post-conditions, and exceptions.

For reference, one solution is around 150 lines long including comments and blank lines (75 "substantive" lines). However this is not a requirement, your code may be as long/short as you need it to be. But if you find yourself going excessively long, you may want to take a step back and reconsider your approach.