

CS 143, Java Programming II

Programming Assignment #3: Evil Hangman

thanks to Keith Schwarz from Stanford for this project idea!

This program focuses on programming with Java Collections classes. Turn in a file named `HangmanManager.java`. You will need support files `HangmanMain.java`, `dictionary.txt`, and other input files from canvas; place them in the same folder as your project.

The Game of Hangman:

In a normal game of hangman, the computer picks a word that the user is supposed to guess. The user then 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_%28game%29.

```
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? 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
```

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 (`dictionary2.txt`):

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 at left, 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 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]

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!

Implementation Details:

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.

Your class should have the following public 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 Set<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.

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

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 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 primitive `char` values.

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.

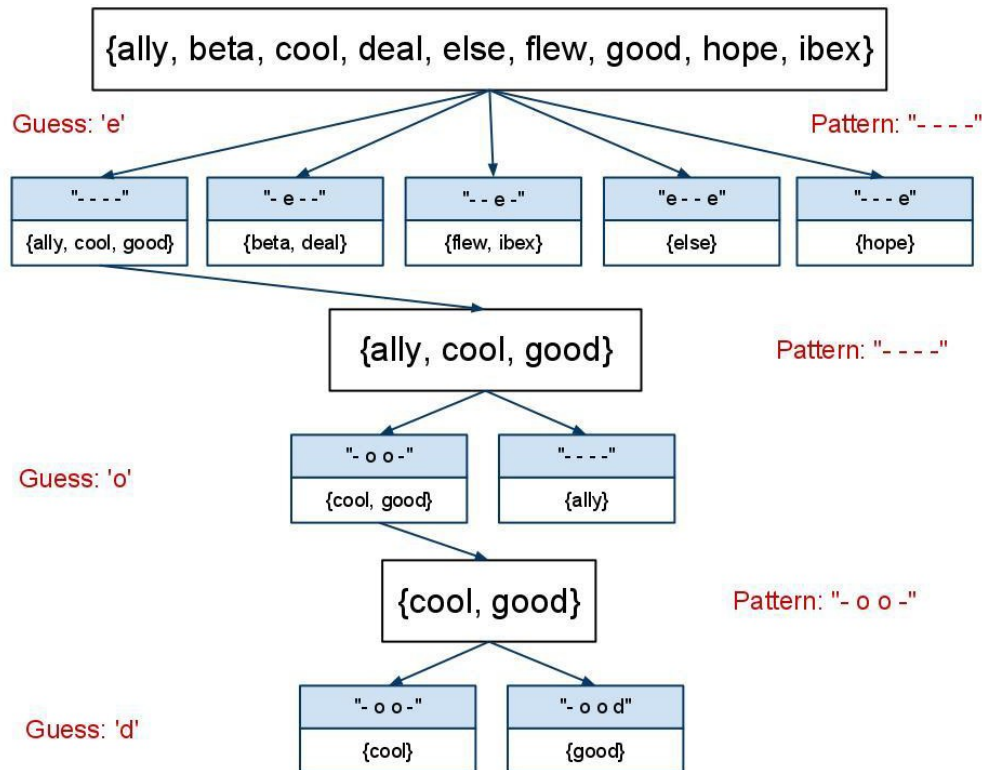
Use the `TreeSet` and `TreeMap` implementations for all of your sets and maps.

Development Strategy and Hints:

The hardest method is **record**, so write it last. There is a more detailed explanation of the record method below.

As a class we will create a simple test client to verify behavior as you add it. For example, we will write code to produce a Hangman-style clue (with dashes for letters that have not been guessed) and verify that it works in our 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. 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. Here is a diagram showing the decisions made by the computer in the scenario described on the first page:



Evil Hangman Constructor Detail

```
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:

1. The tree set of words(`Set<String>`)-initially all words from the dictionary that are of the given length
2. The initial pattern(`String`) would initially be all dashes of the given length like “ - - - - ”
3. The number of guesses left (integer)would initially be set to max
4. The set of characters that contains the letter guesses the player has made (`Set<Character>`)

Below is the output from running `HangmanMain.java` with the following two changes:

```
public static final String DICTIONARY_FILE = "dictionary2.txt";
public static final boolean DEBUG = true; // show words left
```

Welcome to the cse143 hangman game.

What length word do you want to use? 4 (int length)

How many wrong answers allowed? 7 (int max)

guesses : 7 (7 is returned from int guessesLeft())

9 words left: [ally, beta, cool, deal, else, flew, good, hope, ibex] (Set<String> initially created from List<String>

dictionary – these 9 words are returned from Set<String> words())

guessed : [] (guesses made returned from Set<Character> guesses() – initially the set of characters will be empty)

current : ---- (pattern returned from String pattern())

Your guess? e (passed to the record(char guess) method – and guess added to the Set<Character>))

Evil Hangman Record Method Detail

public int record(char guess)

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 patterns with the set of words that have each pattern. If there is a tie (two of the word sets 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 Set<String> words() for the next round. 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 Set<String> words() 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 Set<String> words() used by the HangmanManager.

The record 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 meaning it would be returned by the public Set<Character> guesses())

Example Run:

```
>> 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
```

(e is passed to the record(char guess) method)

The record method looks at each word in the `Set<String> words()` and creates a pattern with the letter e in it that matches that particular word. The options would fall into five map entries:

Pattern(key)	Set (values)
----	[ally, cool, good]
-e--	[beta, deal]
--e-	[flew, ibex]
---e	[hope]
e--e	[else]

The record method then choses the pattern that has the most words in it's set. In this case, the words and pattern to be considered moving forward would be

---- [ally, cool, good]

Since the program did not reveal any letters, the letter guess of 'e' is marked as a wrong guess and the record method decreases the number of guesses left to 6 and adds 'e' to the set of guesses made (`Set<Character> guesses()`). The pattern moving forward ends up staying the same ----.

```
>> Sorry, there are no e's
>> guesses : 6
>> guessed : [e]
>> current : ----
>> Your guess? 0
```

(o is passed to the `record(char guess)` method)

The record method looks at each word in the `Set<String> words()`- [ally, cool, good] and creates a pattern with the letter o in it that matches that particular word. The options would fall into two map entries:

Pattern(key)	Set (values)
----	[ally]
-oo-	[cool, good]

The record method then choses the pattern that has the most words in it's set. In this case, the words and pattern to be considered moving forward would be

-oo- [cool, good]

Since the program did reveal letters, the letter guess of 'o' is not marked as a wrong guess and the record method makes no changes to the number of guesses left but adds 'o' to the set of guesses made (`Set<Character> guesses()`). The pattern moving forward ends up changing to -oo-.

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

(d is passed to the `record(char guess)` method)

The record method looks at each word in the `Set<String> words()`- `[cool, good]` and creates a pattern with the letter d in it that matches that particular word. The options would fall into two map entries:

Pattern(key)	Set (values)
-oo-	[cool]
-ood	[good]

There is now a tie between the sets with the most words. The record method breaks the tie by choosing the pattern -oo- which comes first in the map's keyset. The word and pattern to be considered moving forward would be

-oo- [cool]

Since the program did not reveal letters, the letter guess of 'd' is marked as a wrong guess and the record method decreases the number of guesses left to 5 and adds 'd' to the set of guesses made (`Set<Character> guesses()`). The pattern moving forward does not change.

```
>> Sorry, there are no d's
>> guesses : 5
>> guessed : [d, e, o]
>> current : -oo-
>> Your guess? c
```

(c is passed to the `record(char guess)` method)

The record method looks at the only word in the `Set<String> words()`- `[cool]` and creates a pattern with the letter c in it that matches the last word left. There would be one map entry:

Pattern(key)	Set (values)
coo-	[cool]

Since the program did reveal letters, the letter guess of 'c' is not marked as a wrong guess and the record method makes no changes to the number of guesses left but adds 'c' to the set of guesses made (`Set<Character> guesses()`). The pattern moving forward ends up changing to coo-.

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

(l is passed to the `record(char guess)` method)

The record method looks at the only word in the `Set<String> words()`- `[cool]` and creates a pattern that matches it `[cool]`. Since the program did reveal letters, the letter guess of 'l' is not marked as a wrong guess and the record method makes no changes to the number of guesses left but adds 'l' to the set of guesses made (`Set<Character> guesses()`). The pattern moving forward ends up changing to cool. The `HangmanMain.java` class determines the game is over since there are no dashes left in the pattern and displays the following:

```
>> Yes, there is one l
>> answer = cool
>> You beat me
```

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 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.

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 values and exceptions.