HW4: HangmanManager (due Thursday, April 30, 2015 11:30pm)

This assignment focuses on using the built in Java `Map` and `Set` collctions. Turn in the following files using the link on the course website:

Thanks to Keith Schwarz for this assignment!

- `HangmanManager.java` – A class that manages a game of Hangman.

You will need the support files `HangmanMain.java`, `dictionary.txt`, and `tiny-dictionary.txt`; place these in the same folder as your program or project. You should not modify the provided files. The code you submit must work properly with the unmodified versions.

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_(game)

In our (EVIL!) 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.

Example Game of Evil Hangman

```
>> 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 : -oo-
>> Your guess? d
>> Sorry, there are no d's
>> guesses : 5
>> guessed : [d, e, o]
>> current : -oo-
>> Your guess? c
>> Yes, there is one c
>> guesses : 5
>> guessed : [c, d, e, o]
>> current : coo-
>> Your guess? l
>> Yes, there is one l
>> answer = cool
>> You beat me
```

For example, suppose that the computer knows the words in `tiny-dictionary.txt`:

`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**
`-oo-`	`[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 computer removes "good" from its consideration (because the family with "cool" is alphabetically before the family with "good") and is now locked into using "cool" as the answer.

We have provided you with 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. In this assignment, you will write a class `HangmanManager` that will manage the state of a game of hangman.

HangmanManager

`HangmanManager` should have the following constructor:

`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 of the given length. You should throw an `IllegalArgumentException` if length is less than 1 or if max is less than 0. You may assume that the list of words passed to your constructor will be a non-empty list of non-empty strings composed entirely of lowercase letters.

`HangmanManager` 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. 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)
```

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 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. You may assume that all guesses passed to record are lowercase letters.

When you are changing patterns, you should build up a new `String`.

Implementation Details

Your program should exactly reproduce the format and general behavior demonstrated on the output comparison tool and 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 inefficiency. For example, regular expressions are overkill for this problem, and, while there is a `toCharArray` method, you would lose style points for using it because it creates an unnecessary extra structure.

`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. Sometimes, the player guesses a letter that is in the word (a correct guess), and others, they guess a letter that isn't in the word (a wrong guess). Note that the number of correct guesses the player makes is indefinite, because they don't count against them.

`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.

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.

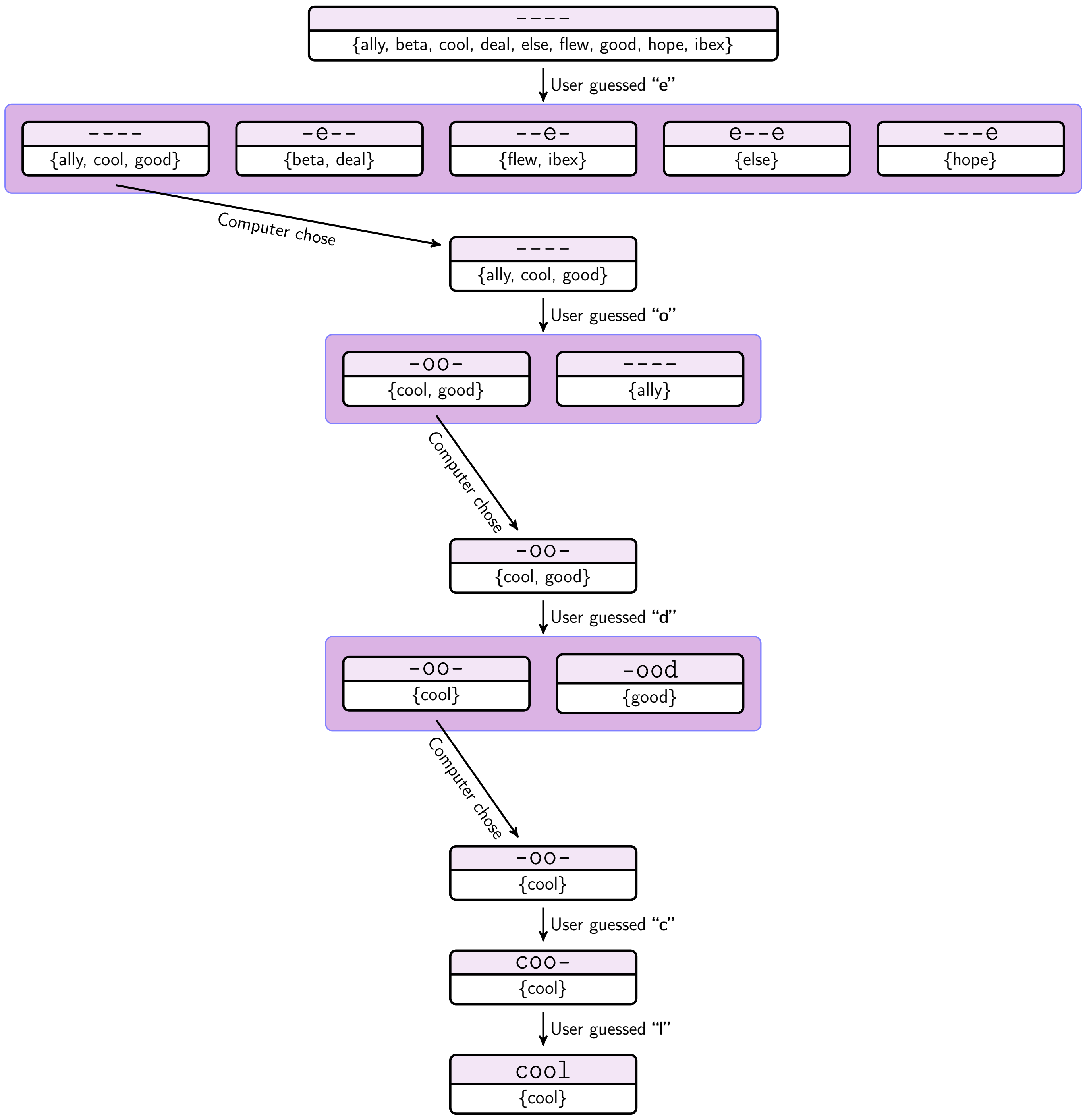
Different words go in different sets because they have different patterns. Once you have processed all of the words, you go through the different sets and find the one with the most words. That becomes the new set used by the HangmanManager.

Development Strategy and Hints

The hardest method is `record`, so 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. 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 `tiny-dictionary.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. Here is a diagram showing the decisions made by the computer in the scenario described on the first page:



Style Guidelines and Grading

Unless otherwise specified, your solution should use only material covered so far. Part of your grade will come from appropriately utilizing maps and sets as described previously.

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, 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.

Java Style Guidelines

Appropriately use control structures like loops and if/else statements. Avoid redundancy using techniques such as methods, loops, and factoring common code out of if/else statements. Properly use indentation, good variable names, and types. Do not have any lines of code longer than 80 characters.

Commenting

You should comment your code with a heading at the top of your class with your name, section, and a description of the overall program. All method headers should be commented as well as all complex sections of code. Make sure you describe complex methods inside methods. Comments should explain each method's behavior, parameters, return values, and assumptions made by your code, as appropriate. The `ArrayIntList` class from lecture provides a good example of the kind of documentation we expect you to include. You do not have to use the pre/post format, but you must include the equivalent information–including the type of exception thrown if a precondition is violated. Write descriptive comments that explain error cases, and details of the behavior that would be important to the client. Your comments should be written in your own words and not taken verbatim from this document.