Lab: Word Games

Style guide rules must be followed. Note that `main` method must be in its own class and that class must be called `Main.` All other style guide rules must also be followed.

Background: String Theory

- Strings are objects that represent character sequences.
- Enclosing text in quotes creates new instances.
- Use escape sequences to create strings containing ", \, or newline characters.
- Different instances may contain the same characters, so don't compare with ==.
- Strings are immutable—no methods will change the sequence of characters.
- "ab" + "cd" will create the new string "abcd".

Here are some messages that the `String` class supports.

```
• boolean equals(Object other)
• int compareTo(Object other)       // return value < 0 if this is less than other
                                    // return value = 0 if this is equal to other
                                    // return value > 0 if this is greater than other
• int length()
• String substring(int from, int to)    // returns the substring beginning at from
                                        // and ending at to-1
• String substring(int from)            // returns substring(from, length())
• int indexOf(String s)                 // returns the index of the first occurrence of s;
                                        // returns -1 if not found
```

Background: WordGameDisplay

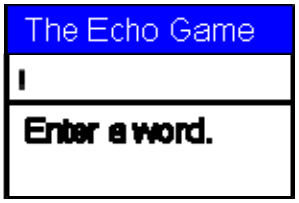
For this lab, you will need to download WordGameDisplay.java , WordGameMonster.java, dictionary.txt, and words.txt to a new directory. The `WordGameMonster` class is a tester for your code and must not be modified. The `WordGameDisplay` class will be our user interface for simple word games. You should *not* need to modify it. Here are the messages it supports.

```
void setTitle(String title)
String getGuess()
void setText(String text)
Iterator<String> loadWords(String fileName)
```

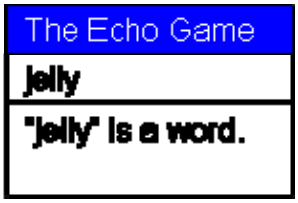
Exercise 1: The Echo Game ... Echo Game ... Echo Game ...

Let's try using a `WordGameDisplay`. Create a class called `WordGame`. In its constructor, create a new `WordGameDisplay` and store it in an instance variable.

Now write a method called `echo`. Calling this method should set the title of the display window to "The Echo Game" and should prompt the user to enter a word. When the user has typed a word and hit enter, the window should "echo" back what the user typed, as shown below. (Yes, the word must appear in quotes.)



Before entering a word

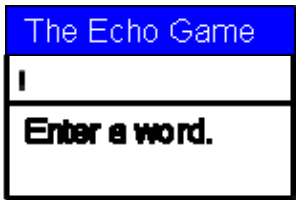


After entering a word

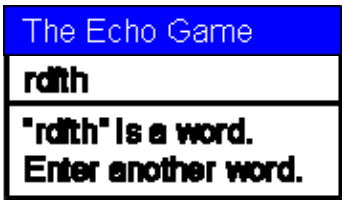
Make sure this fun game works correctly before going on. Then go ahead and make your game even *more* fun (is that even possible?) by making the game repeat *forever*, prompting the user for a new word each time. You can use the following construct to program this infinite loop.

```
while (true)
{
        ... your lousy code ...
}
```

Your game should now appear as follows. (Yes, the prompt for another word must appear on a new line.)



Before entering the first word



After entering each word

Exercise 2: Is "dictionary" in the Dictionary?

Answers to the questions posed must be in your notebook and are worth 5 points.

Now, as you recall from English class, "rdfth" is not actually a word, and therefore your program is lying. We can fix this by searching for the user's word in our "dictionary" file, dictionary.txt, which contains a list of words in alphabetical order. Download dictionary.txt, if you haven't already. Open the file in a text editor and take a look at it. It contains only 20 words. In `WordGame`'s constructor, use your `WordGameDisplay` to load these words into an `ArrayList`, and store it in an instance variable.

Suppose the user types "dictionary", and we wish to check if it's in the dictionary. We could do this by comparing "dictionary" to "apple", then to "bear", "cat", etc., until we reach "zoo", and can safely declare that "dictionary" does not appear in the list. This algorithm is called a *sequential search*. If there are *n* words in our dictionary, what will be the running time of the sequential search algorithm? Answer this question in your notebook before going on. You must justify your answer.

It would be much better to use the binary search algorithm, which more closely resembles the way you would look for a word in a dictionary.

At first, we will consider the range from the first word (index 0) to the last one (index 19), and compare our word "dictionary" to the word in the middle of the range. The middle of the range can be calculated using integer arithmetic to be:

$$\text{middle} = (\text{last} - \text{first}) / 2$$
$$\text{middle} = (19 - 0)/2 = 9$$

The word at index 9 is "join", and if we happened to be looking for "join" we would be done. But we are not looking for "join", but rather "dictionary" which comes before "join" in the list. Remember that the list is in alphabetical order – if it is not, then binary search won't work!

Since "dictionary" comes before "join", which is in the middle of our list, we can ignore all of the words after "join" **and** we can ignore the word "join". Our search range is now reduced to the range [0,middle). The first word of our reduced list is at index 0, and the last word is at index 8.

We again calculate the middle of our reduced list:

$$\text{middle} = \text{first} + (\text{last} - \text{first})/2$$
$$\text{middle} = 0 + (8 - 0)/2 = 4$$

The word at index 4 is, in fact "dictionary", and we can conclude that "dictionary" is in fact in our list. We only had to check our list twice!

Suppose that instead of “dictionary”, we were looking for “nonsense”. Just as before, we look at the middle of the list (index 9) and find the word “join”. It is not the word we are looking for, and our word comes after “join”. We again reduce the range of our search to the second half of the list (throwing out “join”). The reduced range is then [10,19]. Calculating the middle of this reduced list:

middle = 10 + (last – first)/2
middle = 10 + (19 – 10)/2 = 14

The word at index 14 in our list is “pass”, and it comes after “nonsense”. We can conclude then that our word, if it is in the list, lies between index 10 and index 13. Reducing our list once more to [10,13] and calculating the middle:

middle = 10 + (13 – 10)/2 = 11

The word at index 13 is “many” and it comes before “nonsense” so we can reduce the range to [12,13]. The middle of this range is 12. The word at index 12 is “none” and it comes before “nonsense”. Since the word at index 12 comes **before** “nonsense”, we calculate that the new range is [12,11], which is indeed nonsense and we conclude that “nonsense” is not in our list. Whenever the word is not in the list, we will ultimately end up with a search range where the beginning of the range is after the end of the range.

Let’s look at the calculations that would occur if the word we seek is “going”.

- probe the middle of the list (index 9), which gives the word “join”
- restrict the range to [0,8], and calculate a new middle index of 4
- probe the middle (index 4) which gives the word “dictionary”
- restrict the range to [5,8] , and calculate a new middle of 6
- probe the middle (index 6) which gives the word “going” and we are done!

If there are *n* words in our dictionary, what will be the running time of the binary search algorithm? In what situation(s) would it make more sense to use the sequential search algorithm? Answer these questions in your notebook. You must justify your answer.

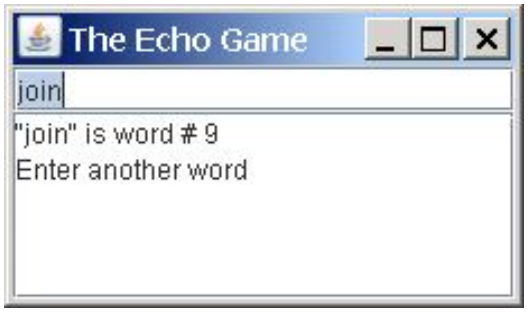
Add a method called `dictionaryIndex` to your `WordGames` class, which should take in a `String` and use the binary search algorithm to return the index of that word in the dictionary. If the word is not in the dictionary, `dictionaryIndex` should return -1. For example, `dictionaryIndex("instance")` should return 8 while `dictionaryIndex("lemonade")` should return -1.

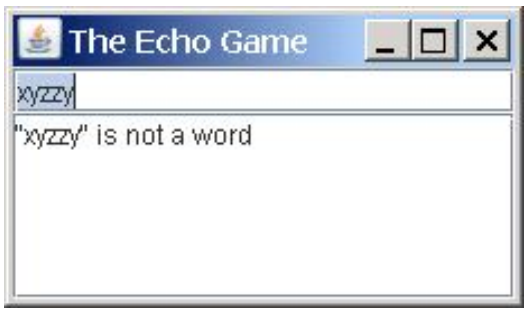
Hint #1: You'll need to keep track of what range of the dictionary you're currently looking in—beginning and end.

Hint #2: You'll need a loop—either a `while` loop or recursion. If you write it recursively, you'll need a helper method to take in the range currently being considered.

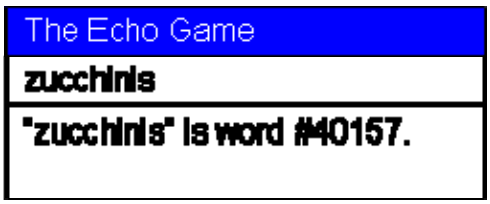
Hint #3: What will the "range" look like when there are no words left in it?

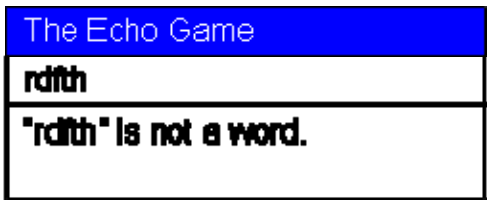
Now use your `dictionaryIndex` method in your "Echo Game" so that it behaves as follows:





After you are convinced your search algorithm works, change the file from "dictionary.txt" to "words.txt". Use your `dictionaryIndex` method in your "Echo Game" so that it behaves as follows:





Your English teacher will be very proud.

Exercise 3: Random Words Fruitlessly Egotism Candidacies

Write a method called `getRandomWord` that returns a random word from the dictionary. It may help you to know that the following mysterious code will assign `num` to be a random integer from 0 to 9.

```
int num = (int)(10 * Math.random());
```

Next, use the method you just wrote to write a second method also called `getRandomWord` that returns a random word of a particular length. (If the random word doesn't match the desired length, simply pick again.) Test that this method works correctly before you go on.

Why is it OK for both methods to have the same name? A complete answer must be in your notebook.

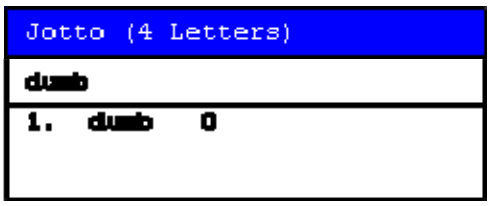
Exercise 4: commonLetters("common", "letters") → 0

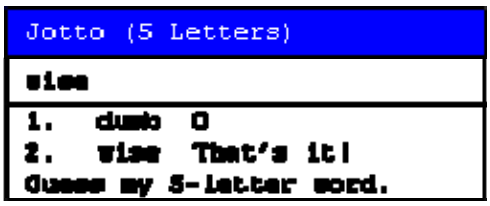
Write a method called `commonLetters` that takes in two words and returns the number of letters they have in common. "bee" and "ebb" have two letters in common, as do "bee" and "eye". Do you see why? Think very carefully about how you are recognizing common letters before you begin to program. Be sure to test this code well! (Hint: Think recursively.)

Before going on, test your code using the WordGameMonster. Your code must pass the WordGameMonster before you may proceed.

Exercise 5: Domo Arigato, Mr. Jotto

In the game "Jotto", the computer thinks of a secret *n*-letter word, and the user tries to guess it. The computer responds by telling the user how many letters the user's guess has in common with the secret word, as shown in the sample game below.





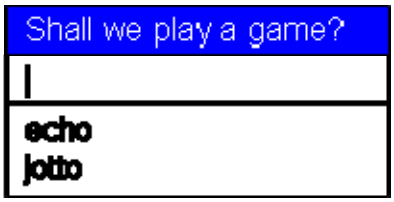
Write a method `jotto` that plays the game of Jotto as shown above. Your game should start with 2-letter words. When the user correctly guesses the computer's secret 2-letter word, the computer should then choose a 3-letter word, and so on. The computer must enforce that the user's guess is a real word of the required length. The user may type "pass" to view the computer's secret word and guess a new one of the same length.

Start simple. First, have the `jotto` method use a `while(true)` loop to allow the user to repeatedly guess a word (without scoring it or providing any feedback), and show the history of all words guessed so far. (You might consider using the escape sequence `\t` to include tab characters.)

Next focus on playing a single round (one secret word) before you progress to choosing a second secret word, etc.

A Change of Menu

Implement a `menu` method, which should ask the user what game to play, and then begin playing that game.



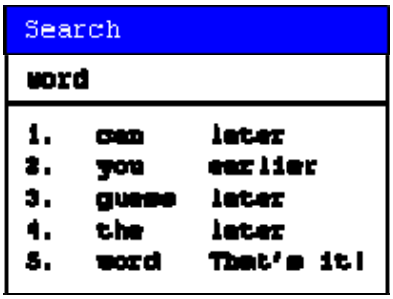
If you finish early…..

Make Your Favorite Word Games

Program any or all of the following word games. Make sure to add your new games to your word game menu.

jumble
Like Jotto, pick a random word, starting with length 3, and pick increasingly longer words as the user guesses correctly. In each round, scramble the secret word, and display it for the user, who gets just one try to guess the secret word.

search
Pick a random secret word of any length. After each guess, tell the user whether the secret word appears "earler" or "later" than it in alphabetical order.



subwords
Pick a random word of maybe 4 or 5 letters (or just a random 4 or 5 letters), and present this to the user. Your program should also find all words that can be made from at least 3 of those letters, and keep track of which of these the user has entered. For example, if the random word is "code", then your program should tell the user there are 3 words that can be made from these letters, and the user will try to guess these ("cod", "doe", "ode"). Ideally, your game should (1) display all the words that the user has found, (2) show the user how many words have not yet been found, and (3) allow the user to give up and see all the words they didn't find.

hangman
Pick a random word, and display a blank for each letter in it. Have the user guess single letters, and update the display to show where these occur in the word. Also show a list of the guessed letters that did not appear in the word.

some other word game
Implement your favorite word game or make up a new one!