Hangman

Thursday, October 25, 6:00 – 7:30PM

Peter Maldonado and Olivia Gregory

Overview

- Review Lecture Material
  - Characters
  - Strings
- Assignment Overview
  - Milestones/breakdown of tasks
  - Some useful upcoming topics
  - General suggestions and reminders
- Q&A

Lecture Review

Characters

```
char ch = 'a';
// need to store return value
ch = Character.toUpperCase(ch);
// converting a char to a string
String str = "" + ch;
```

Useful methods in the Character Class

`static boolean isDigit(char ch)` Determines if the specified character is a digit.
`static boolean isLetter(char ch)` Determines if the specified character is a letter.
`static boolean isLetterOrDigit(char ch)` Determines if the specified character is a letter or a digit.
`static boolean isLowerCase(char ch)` Determines if the specified character is a lowercase letter.
`static boolean isUpperCase(char ch)` Determines if the specified character is an uppercase letter.
`static boolean isWhitespace(char ch)` Determines if the specified character is **whitespace** (spaces and tabs).
`static char toLowerCase(char ch)` Converts **`ch`** to its lowercase equivalent, if any. If not, **`ch`** is returned unchanged.
`static char toUpperCase(char ch)` Converts **`ch`** to its uppercase equivalent, if any. If not, **`ch`** is returned unchanged.

Using portions of slides by Eric Roberts

Comparing Characters

- Write a program that…
  - …prompts the user for 2 words
  - …prints out "The first letters match!" if the first letters of the two words are the same and "The first letters differ" if the first letters are not the same
    - Case-insensitive (so "CS106A" and "cs106a" should match)

Comparing Characters - Solution

```
String first = readLine("Enter a word: ");
String second = readLine("Enter a word: ");
```

Comparing Characters - Solution

```
String first = readLine("Enter a word: ");
String second = readLine("Enter a word: ");
if (Character.toLowerCase(first.charAt(0)) ==
     Character.toLowerCase(second.charAt(0))) {
    println("The first letters match!");
} else {
    println("The first letters differ.");
}
```

Comparing Characters - Solution

```
String first = readLine("Enter a word: ");
String second = readLine("Enter a word: ");
if (Character.toLowerCase(first.charAt(0)) ==
     Character.toLowerCase(second.charAt(0))) {
    println("The first letters match!");
} else {
    println("The first letters differ.");
}
```

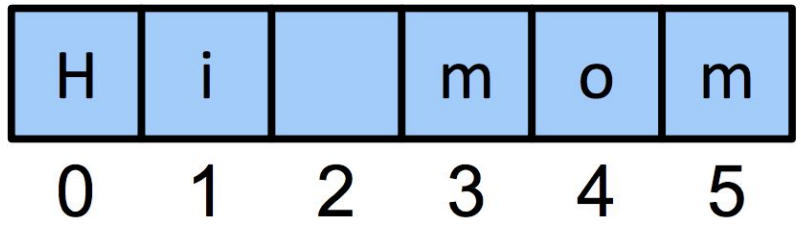
What if the user enters an empty string?

Comparing Characters - Solution

```
String first = readLine("Enter a word: ");
String second = readLine("Enter a word: ");
if (first.length() == 0 || second.length() == 0) {
    println("Empty string");
} else if (Character.toLowerCase(first.charAt(0)) ==
           Character.toLowerCase(second.charAt(0))) {
    println("The first letters match!");
} else {
    println("The first letters differ.");
}
```

Strings

```
String s = "Hi mom";   // ordered characters
```




```
// need to store value of s.toUpperCase()
s = s.toUpperCase();
println(s);              // prints "HI MOM"
```

Useful methods in the String Class

`int length()` Returns the length of the string
`char charAt(int index)` Returns the character at the specified index. Note: Strings indexed starting at 0.
`String substring(int p1, int p2)` Returns the substring beginning at **`p1`** and extending up to but not including **`p2`**
`String substring(int p1)` Returns substring beginning at **`p1`** and extending through end of string.
`boolean equals(String s2)` Returns true if string **`s2`** is equal to the receiver string. This is case sensitive.
`int compareTo(String s2)` Returns integer whose sign indicates how strings compare in lexicographic order
`int indexOf(char ch)` *or* **`int indexOf(String s)`** Returns index of first occurrence of the character or the string, or -1 if not found
`String toLowerCase()` *or* **`String toUpperCase()`** Returns a lowercase or uppercase version of the receiver string

Looping over a String

Canonical “loop over the characters in a string” loop

```
for (int i = 0; i < string.length(); i++) {
    char ch = string.charAt(i);
    /* … process ch … */
}
```

*[str].length() returns length not final index

Comparing Strings

```
String s1 = “racecar”;
String s2 = reverseString(s1);

// How do we check equality?
```

Comparing Strings

```
String s1 = "racecar";
String s2 = reverseString(s1);

// How do we check equality?
```

```
if (s1 == s2) {
    …
}
```

OR

```
if (s2.equals(s1)) {
    …
}
```

Comparing Strings

```
String s1 = "racecar";
String s2 = reverseString(s1);

// How do we check equality?
if (s1 == s2) {
    …
}
```

DON'T DO THIS

String References vs. Literals

```
String s1 = "racecar";
String s2 = reverseString(s1);

// How do we check equality?
if (s1 == s2) {
    …  // Compares reference address
}
```

```
if (s2.equals(s1)) {
    … // Compares string literal
}
```

Stack	reference address
s1	EE40
s2	EE48

Heap	string literal
EE40	"racecar"
EE48	"racecar"

String References vs. Literals

```
String school = "Harvard";
println(school + " is a top university..")
fix(school);
println(school + " is not as good as Stanford!")

// What is printed?
private void fix(String str) {
    str = "Stanford of the East";
}
```

String References vs. Literals

Stack (run)	reference address
school	AC36

Heap	string literal
AC36	"Harvard"
AC4D	"Stanford of the East"

```
String school = "Harvard";
println(school + " is a top university..") // Harvard is a top university..
fix(school);
println(school + " is not as good as Stanford!") // Harvard is not as good as
                                                 // Stanford!

// Because strings are immutable they behave like primitives in methods!
private void fix(String str) {
    str = "Stanford of the East";
}
```

Stack (fix)	reference address
str	AC4D

String References vs. Literals

Stack (run)	reference address
school	~~AC36~~ AC4D

Heap	string literal
AC36	"Harvard"
AC4D	"Stanford of the East"

```
String school = "Harvard";
println(school + " is a top university..") // Harvard is a top university..
school = fix(school);
println(school + " is not as good as Stanford!") // Stanford of the East is
                                                 // not as good as Stanford!
// We must return a new string if we want to modify school in run()
private String fix(String str) {
    str = "Stanford of the East";
    return str;
}
```

Stack (fix)	reference address
str	AC4D

Searching Strings

- You can use the `indexOf` method to search a string:

```
int index = str.indexOf(pattern);
```

- `indexOf` returns the start index of the first occurrence of the pattern if the pattern exists in the string. Otherwise, if returns -1
- Overloaded so that `pattern` can be either a **`String`** or a **`char`**!

```
int index = "hello".indexOf("el");          // 1
int index = "hello".indexOf('e');       // 1 (overloaded to work with chars!)
int notFound = "cs106a".indexOf("b");       // -1
```

Building Strings

- 1. Use substrings – smaller pieces of strings

  OR

- 2. Make new string and build over time

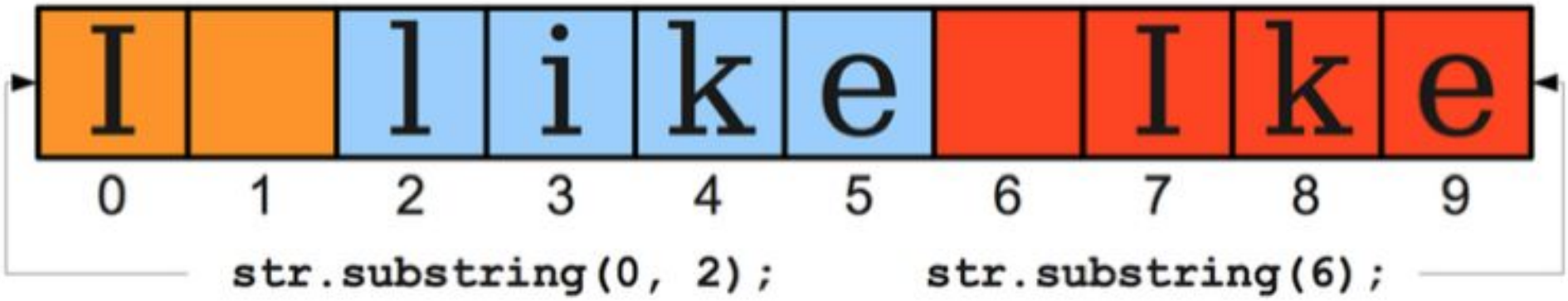
1. Substrings

- To get all of the characters in the range [start, stop), use

```
str.substring(start, stop);
```

- To get all of the characters from some specified point forward, use

```
str.substring(start);
```



Building a New String

- Start with an empty string and build up a new string
- Iterate through the old string
- Use Character methods at each position to decide what to concatenate to the new string
- See this week's section handout for examples

String Summary: Strings are...

- objects that have methods (`length()`, `charAt()`, `equals()`, `indexOf()...`)
- zero-indexed list of `chars`
- **immutable!**
    - but you can concatenate them, get substrings from them, search them, compare them
    - ...using **methods** and the canonical **new string + reassignment to old** pattern

Classes Example

```
public void run() {
    BankAccount brahm = new BankAccount(5);
    BankAccount mehran = new BankAccount(1000);

    int curBal = mehran.checkBalance();
    mehran.setBalance(curBal + 50);

    brahm.bankError(); // can this happen?

    // what prints out?
    println(brahm.checkBalance()); // 5!
    println(mehran.checkBalance()); // 1050!

    // Brahm has 5 because banks are secure! NO!
    // It's because we can't use access a
    // private method outside that class
}
```

```
public Class BankAccount {
    private int balance;

    public BankAccount(int initBalance) {
        balance = initBalance;
    }

    // "getter"
    public int checkBalance() {
        return balance;
    }

    // "setter"
    public void setBalance(int newBal) {
        balance = newBal;
    }

    private void bankError() {
        balance -= 10;
    }
}
```

Assignment 4 - Hangman!

Assignment 4 - Hangman

- Due Monday, Nov 5
- String processing
- Pair assignment (optional)
- We suggest approaching this assignment in stages

Part I: Playing a Console-Based Game

Part I: Console Game

- Choose a **random word** (using HangmanLexicon)
- Display a "**hint**" (initially "- - - - - - - - - -")
- Get **guesses** from the user
- Figure out if a guess is **correct** (letter in the secret word) or **incorrect** (not in secret word)
- **Update** hint
- Keep track of the **number of guesses** the user has left
- Determine when the game has **ended** (no guesses left or they guessed the word)
- **...Repeat**

Game Flow

`String secretWord`	P R O G R A M M E R
`String wordState`	- - - - - - - - - -
`char guess`	r

`String newWordState`	-R--R----R

Hangman Lexicon

- Stub class you can use until you are able to implement part 3
- Beginning of run: create a new **HangmanLexicon** and store it in an **instance variable**
- If you extend the program to allow the user to play multiple games, create HangmanLexicon outside the loop that plays the game repeatedly

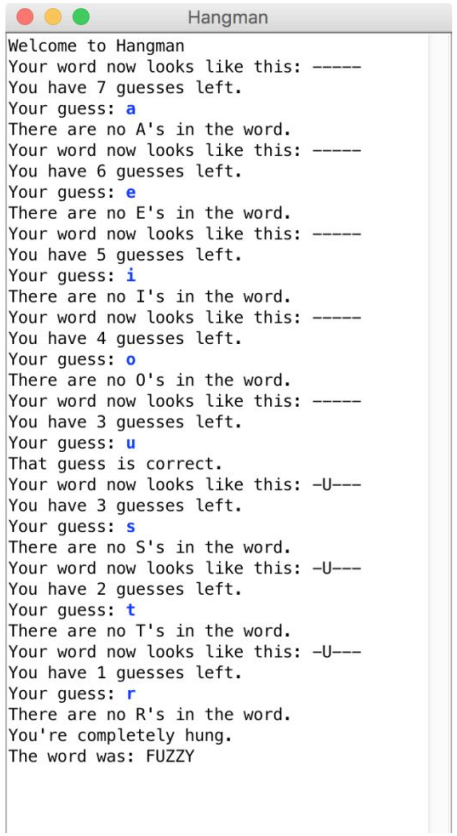
Part I: Console Game - Tips

- Keep track of the user's partially-guessed word (dashes and letters)
- Your program should be **case-insensitive** (**R** and **r** should be the same guess)
  - Guessed letters string should be all upper-case, even when a guess is lower case
- You will have some **fencepost** issues – look at lecture slides for techniques to deal with this
- Watch out for edge case input! (single letter, empty string, etc)

Part I: Console Game - Error Checking

- You'll need to prompt the user to enter guesses
- The user may enter a letter in upper or lower case (hint: the secret words are all upper-case)
- If the user guesses anything other than a single letter, print out an error message and reprompt
- If the user enters the same correct letter more than once, do nothing.
- If the user enters the same *incorrect* letter more than once, it's incorrect again.

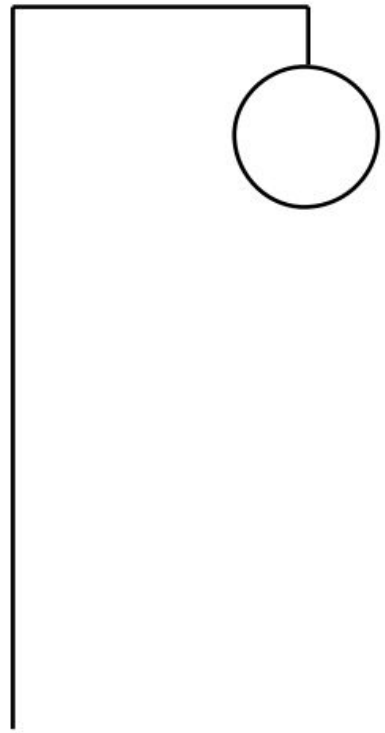
Part I: Console Game – Sample Output

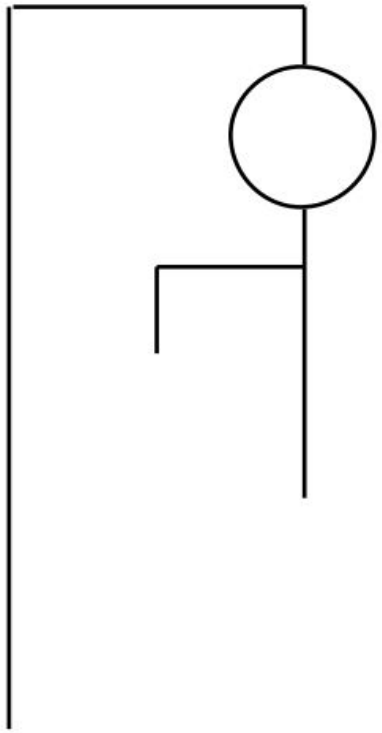


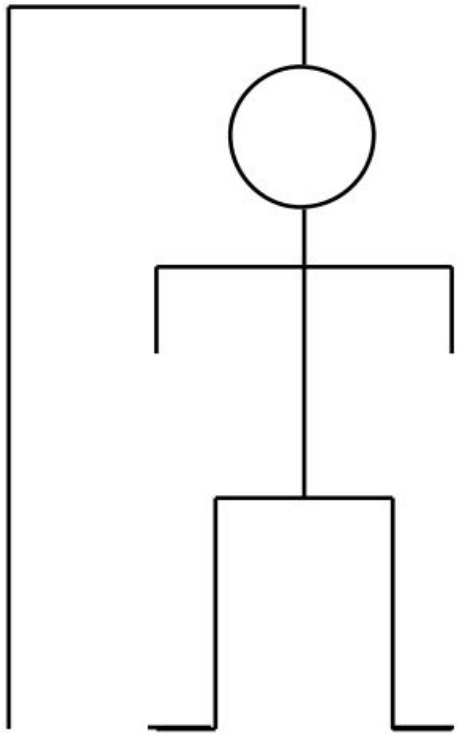
Follow the screenshots to know what your output should look like!

Part II: Adding Graphics

Part II: Hangman Graphics



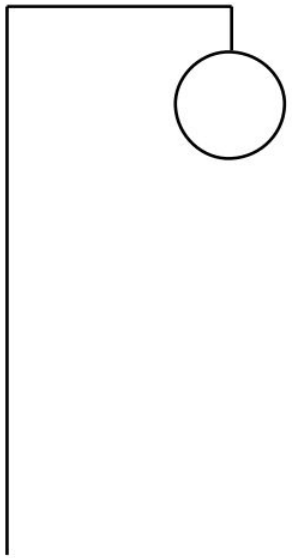




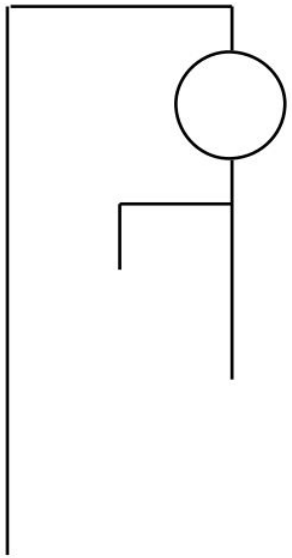
Part II: Hangman Graphics

- Add the canvas instance variable to the window using `init()`
  - Call graphics methods on the canvas object, since console programs don't know how to do graphics tasks! i.e.: `canvas.add(object, x, y);`
- **reset**: Reset canvas whenever you start a game (scaffold already drawn)
- **noteIncorrectGuess**: Add new body part, add letter to list of incorrect guesses at bottom of window
  - For body parts, use the exact order specified in the handout
- **displayWord**: Add current word state
- **DO NOT CHANGE THE INTERFACE** (but you can add any *private members* that you would like)

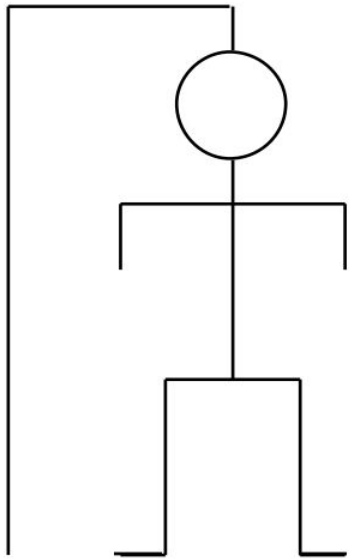
Part II: Add & remove body parts



7 guesses left



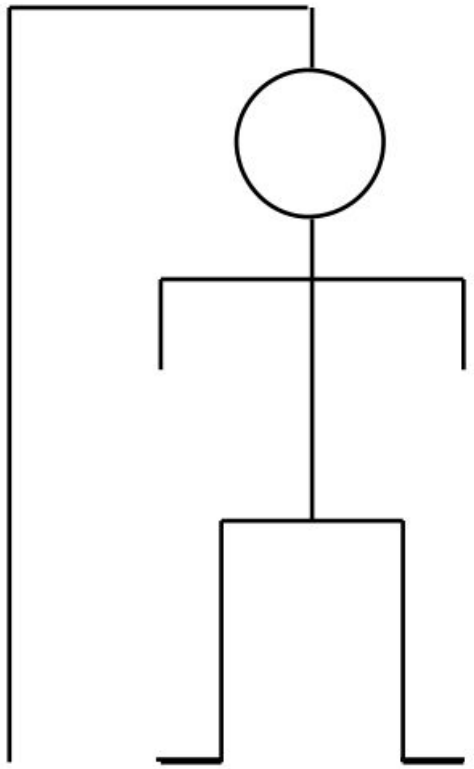
5 guesses left



game over

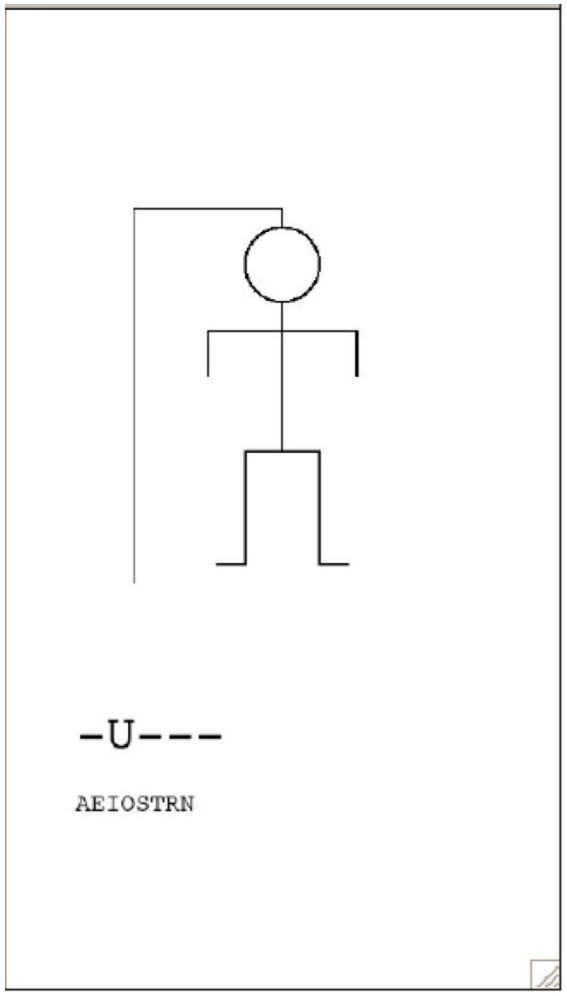
Part II: Add & remove body parts

- Default 8 parts (protip: constant)
- Sizes of the scaffold and body parts are given to you! You will have to do some arithmetic to calculate the coordinates.
- Body should be centered horizontally
- Nothing should go off the screen -- Scaffold should be displayed a bit higher than the center so that there is room underneath for: a label in a large font showing the secret word as it currently stands and a label in a smaller font showing the incorrect guesses.



Part II: Add labels for game state

- Use GLabels to represent current state of guessed word and incorrectly guessed letters
- Update these when the game state changes due to user input



Part III: Reading The Lexicon From a Data File

Part III: Random Word from File

```
private String getRandomWord()
```

- Before starting this milestone, just use the provided "stub" implementation to get one of 10 random words.
- 1. Open the data file **`HangmanLexicon.txt`** using a BufferedReader (at start of program)
- 2. Read the lines from the file into an **`ArrayList`** (at start of program)
- 3. Reimplement **`getRandomWord`** so it uses this ArrayList as the source of the words.

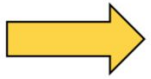
There is also a **`ShorterLexicon.txt`** file you can use for testing/debugging.

(You'll learn about **ArrayLists** and **BufferedReaders** in class Fri/Mon!)

Upcoming Useful Topics

BufferedReader

- Use a `BufferedReader` to read from a file



```
Yesterday, upon the stair,
I met a man who wasn't there
He wasn't there again today
I wish, I wish he'd go away...
- Hughes Mearns, "Antagonish"
```

```
BufferedReader br =
   new BufferedReader(new FileReader("poem.txt"));

String line1 = br.readLine(); // Yesterday, upon the stair,
String line2 = br.readLine(); // I met a man who wasn't there
String line3 = br.readLine();
```

Thanks Chris Piech for some great examples!

BufferedReader

```
try {
    BufferedReader br =
       new BufferedReader(new FileReader("poem.txt"));

    String line1 = br.readLine();  // Yesterday, upon the stair,
    String line2 = br.readLine();  // I met a man who wasn't there
    String line3 = br.readLine();  // He wasn't there again today
    String line4 = br.readLine();  // I wish, I wish he'd go away...
    String line5 = br.readLine();  // - Hughes Mearns, "Antagonish"
    String line6 = br.readLine();  // *Returns null*

    br.close();
} catch (IOException e) {
    println("An error occurred: " + e);
}
```

BufferedReader

- To use a method or class that could cause an exception, we have to tell Java to **`try`** its best, even though it might fail.
    - We'll see this most often when dealing with file reading
- If we do run into an exception, we ask Java to **`catch`** that exception

File Concept in One Slide (from Chris Piech)

1. Make a BufferedReader (let's call it br) to open a file for reading

```
BufferedReader br = new BufferedReader(new FileReader("poem.txt"));
```

2. Use br.readLine to get one line from the file (use a loop to repeat until read null for whole file)

```
br.readLine(); // returns the next line, or null
```

3. Both the above operations are "dangerous so we need to use a try/catch block

```
try {
    // live dangerously
} catch (Exception e) {
    // have health insurance
}
```

4. You can either handle the problem or throw a runtime exception

```
throw new RuntimeException("AHHHH!");
```

What's an ArrayList??

- Class representing an ordered, variable size list of data
- Homogeneous (each entry is of the same data type)
- Can store any object!
- (Remember to import `java.util.*;`!)

ArrayList Example

```
ArrayList<String> ourFirstArrayList = new ArrayList<String>();
// Add elements to the right (the 'back')
ourFirstArrayList.add('hello');
ourFirstArrayList.add('world');
// Elements 0-indexed left ('front') to right ('back')
println(ourFirstArrayList.get(0) + " " + ourFirstArrayList.get(1);
// Will get an IndexOutOfBoundsException since size is 2, but we 0-index!
String tmp = ourFirstArrayList.get(ourFirstArrayList.size());
```

Helpful ArrayList Methods

`boolean add(<T> element)` Adds a new element to the end of the **`ArrayList`**; the return value is always **`true`**.
`void add(int index, <T> element)` Inserts a new element into the **`ArrayList`** before the position specified by **`index`**.
`<T> remove(int index)` Removes the element at the specified position and returns that value.
`boolean remove(<T> element)` Removes the first instance of **`element`**, if it appears; returns **`true`** if a match is found.
`void clear()` Removes all elements from the **`ArrayList`**.
`int size()` Returns the number of elements in the **`ArrayList`**.
`<T> get(int index)` Returns the object at the specified index.
`<T> set(int index, <T> value)` Sets the element at the specified index to the new value and returns the old value.
`int indexOf(<T> value)` Returns the index of the first occurrence of the specified value, or −1 if it does not appear.
`boolean contains(<T> value)` Returns **`true`** if the **`ArrayList`** contains the specified value.
`boolean isEmpty()` Returns **`true`** if the **`ArrayList`** contains no elements.

Wrapping Up

Extensions

- Extensions are optional, and you will get a small amount of extra credit if you do them
  - Focus on the main program first, though – extensions won't make up for a broken Hangman!
- If you do extensions, submit two different .java files for the assignment
  - The basic Hangman.java that meets all of the assignment requirements
  - HangmanExtra.java that has your extensions. *In Eclipse, right click on Hangman.java, click Copy, then ctrl+v (paste). In the Name Conflict window that appears, write HangmanExtra and click OK, then make extension edits in the new file. Both files will submit together.*
- In HangmanExtra.java, be sure to comment all of your extensions in the header comment so your SL knows what to look for.
- See the spec for ideas or come up with your own!

Final Tips

- Make sure your program compiles without any errors or warnings
- Follow the spec carefully and make sure your output matches the spec and expected output
- Make sure you properly handle all user input, including faulty/unexpected input
- Use instance variables only where absolutely necessary
- Don't have a method that calls itself
- Go to the LaIR if you get stuck, and incorporate IG feedback!

questions?

fin.