Nick Troccoli
CS 106A

Assignment 3
July 12, 2017

Assignment #3 — Hangman
Due: 11AM PST on Thursday, July 20th

This assignment may be done in pairs (which is optional, not required)

Based on handouts by Mehran Sahami, Eric Roberts and Marty Stepp

For this assignment, your mission is to write a program that plays the game of Hangman. As an assignment, Hangman will give you practice with Strings, file processing, parameters, and return values, while also in itself being a fun console-based game to play. You should implement all required parts of the assignment in the file **`Hangman.java`**.

Note that this assignment may be done in **pairs**, or may be done individually. **You may only pair up with someone in the same section time and location**. If you would like to work with a partner but don't have one, you can try to meet one in your section. If you work as a pair, **comment both members' names** on top of every .java file. **Only one** of you should submit the assignment; do not turn in two copies.

Note that **you should limit yourself to the material covered up until the release of this assignment** (through lecture on Wednesday, July 12th). **You should not use any other material (in particular private instance variables)**. You may, however, use what we learn about graphics programs for optional extensions to this assignment. If you have any questions about what is ok to use, feel free to ask.

Comparing Output**:** each of your programs must *exactly* match the specified output. To check your output, use the Output Comparison tool; see the Assignment 2 handout for more information. You may also directly view sample output files by opening them in Eclipse from inside the `output/` folder, or via the Assignment 3 page of the course website.

The Game of Hangman

Hangman is a single-player game where the player has a finite number of guesses to try and guess all the letters that make up a secret word. After printing an introductory message explaining the game to the player, the computer selects a secret word at random. Then the player does a series of turns. In each turn, the player guesses a letter from A-Z. Incorrect guesses are displayed as an evolving picture of the player being hanged at a gallows. For each incorrect guess, a new part of a stick figure—first the head, then the body, then each arm, each leg, and finally each foot—is added until hanging is complete.

On each turn, the program shows a hint about the secret word. The hint is initially a row of dashes, one for each letter in the secret word. For example, if the secret word is `"HELLO"`, the hint is `"-----"`. If the player's guess is a letter that appears in the secret word, the hint is updated so that all instances of that letter are shown in their correct positions. For example, if the secret word is `"SHELLS"` and the player guesses `"H"`, the hint becomes `"-H----"`. If the player then guesses `"L"`, the hint becomes `"-H-LL-"`.

Note that your program should be case-insensitive; it should accept uppercase or lowercase letters and treat them the same. The game ends when either the user has correctly guessed all the letters in the secret word, or the user has made eight incorrect guesses.

At the end, the program reveals the secret word, indicates whether the player won or lost, and asks if they would like to play again. If the player chooses to play again, another game begins. When the player chooses not to play again, the program prints statistics about all games. Show the total number of games, games won, percentage of games won (a real number), and best game (most guesses remaining at the end of a game).

(partial log of execution)

```
   +------------+          __________
   |            |         /          \
   |         _______      | Help me! |
   |        / o   o \   / \__________/
   |        |   .   |  /
   |        |  ___  |
   |        \_______/
   |            |
   |      \\----+----//
   |            |
   |            |
   |           / \
   |          /   \
   |       __/
   |
---+---
Secret word : --O-------
Your guesses: OAXZKVTQ
Guesses left: 1
Your guess? p
Correct!
...
```

(see expected output logs for complete output)

As in past assignments, we ask you to break apart the overall task into methods; however, since we are now using parameters and return values and writing a much larger program, it is difficult for a new programmer to come up with good decomposition. Therefore, we are going to tell you what methods you should have (and **require these methods**) in your program. You may create additional methods if you like, but you **must** have the methods shown below with *exactly* these names and *exactly* these parameters (no more, no less) and return types. **Do not change the method definitions** in any way, or you will be substantially penalized. The reason we are requiring these methods is to provide practice using **parameters and return values** to communicate between these methods.

```
public void run()
private void intro()
private int playOneGame(String secretWord)
private void displayHangman(int guessCount)
private String createHint(String secretWord, String guessedLetters)
private char readGuess(String guessedLetters)
private String getRandomWord(String filename)
private void stats(int gamesCount, int gamesWon, int best)
```

List of required methods in Hangman (you must write all of these methods as shown)

Eventually, your **`playOneGame`** method must call the methods above to help it solve the overall task of playing the game. Since this is a challenging program, we suggest that you develop it in stages. The following pages outline a series of stages that we strongly recommend you follow in order.

Task 0: Introduction Message

Before we implement the main game, write a method to print the program introduction to the player. Also write an initial version of your `run` method that will simply call `intro` for now, so you can run it and verify that this method has been written properly.

```
private void intro()
```

In this method, you should print the following introductory text that appears at the start of the program. A blank line of output should appear after the text.

```
CS 106A Hangman!
I will think of a random word.
You'll try to guess its letters.
Every time you guess a letter
that isn't in my word, a new body
part of the hanging man appears.
Guess correctly to avoid the gallows!

```

Task 1: A Single Game

Start your development of the program by writing the code to play just a single game, without displaying the hanging man graphic. Your task is to write the following method:

```
private int playOneGame(String secretWord)
```

In this method, you should do all of the work to play a single game of Hangman with the user from start to finish. Your method will be passed as a parameter the secret word for the user to guess. Your method should return the number of guesses the player had remaining at the end of the game, or 0 if the player lost the game.

You should also modify your program's `run` method to simply call `playOneGame` once, for now. Make the secret word always be a particular word of your choice, such as `"PROGRAMMER"`.

```
public void run() {
     playOneGame("PROGRAMMER");
}
```

For now, don't worry about playing multiple games or displaying statistics. You will need code to do tasks such as the following:

- Display a "hint" about the secret word on each turn; initially a string of dashes
- Ask the user to type a valid guess
- Figure out if a guess is correct (in secret word) or incorrect (not in secret word)
- Keep track of the number of guesses remaining
- Determine when the game has ended (when the user guesses all letters in the secret word, or runs out of guesses)

At right is a sample log of execution of this stage of the programming assignment, using "PROGRAMMER" as the secret word (*User input appears bold and blue)*.

When you are ready to test your program, for now you can do so by changing the secret word specified in the `run` method.

During the game, you will need to keep track of the "guessed letters" string, which stores every letter, both correct and incorrect, that the user has guessed. This is useful for checking if a user has already guessed a letter, and in producing the "hint," because it tells you which letters from the secret word should be shown. For example, if the secret word is "PROGRAMMER" and the guessed letters string is "LMPTOI", the hint should be "P-O---MM--".

Your program should be **case-insensitive**. You should accept the user's guesses in either lower or uppercase, even though all letters in the secret word are written in uppercase. The guessed letters string should be in all uppercase.

In this and other parts of the assignment, you may encounter **fencepost** issues like in earlier assignments. Remember the fencepost strategies we saw in class!

```
Secret word : ----------
Your guesses:
Guesses left: 8
Your guess? o
Correct!

Secret word : --O-------
Your guesses: O
Guesses left: 8
Your guess? x
Incorrect.

Secret word : --O-------
Your guesses: OX
Guesses left: 7
Your guess? a
Correct!

Secret word : --O--A----
Your guesses: OXA
Guesses left: 7
Your guess? o
You already guessed that letter.
Your guess? what do you mean?
Type a single letter from A-Z.
Your guess? r
Correct!

Secret word : -RO-RA---R
Your guesses: OXAR
Guesses left: 7
Your guess?

... (output shortened for space;
    see logs on course web site)

Secret word : -ROGRAMMER
Your guesses: OXARMBKEGTY
Guesses left: 2
Your guess? p
Correct!
You win! My word was "PROGRAMMER".
```

Note that if you wrote the entirety of the code for playing one game in `playOneGame`, the method would be very long and poorly decomposed. To help you break apart this task, we **require** that you write and use the following additional methods. The idea is that your `playOneGame` method should call these methods as part of its overall task.

`private String` **`createHint`**`(String secretWord, String guessedLetters)`

In this method, you should create and return a hint string. Your method should accept two parameters: the secret word that the user is trying to guess, and the set of letters that have already been guessed. For example, if the user has guessed E, T, O, S, and X, you will be passed the guessed letters string "ETOSX". Your task is to create a version of the secret word that reveals any guessed letters but shows dashes in place of all other letters. For example, if the secret word is "STARTED" and the guessed letters are "ETOSX", you should return "ST--TE-". You may assume that both parameters are entirely in uppercase; the guessed letters string could be empty if the user has not guessed any letters yet. Note that this method should not print anything to the console.

Here are some example calls to this method, and their expected results:

- `createHint("STARTED", "ETOSX")` should return `"ST--TE-"`
- `createHint("PROGRAMMER", "RMPAO")` should return `"PRO-RAMM-R"`
- `createHint("COMPUTER", "")` should return `"--------"`

`private char` **`readGuess`**`(String guessedLetters)`

In this method, you should prompt the user to type a single letter to guess, and return the letter typed as an uppercase char. Your method should accept a string as a parameter representing all letters that have already been guessed; for example, if the user has guessed T, O, S, and X, you will be passed `"TOSX"`. If the user has not guessed any letters yet, the string will be empty. You should re-prompt the user until they type a string that is a single letter from A-Z, case-insensitive, that has not been guessed before.

The following is a log of console output of one call of this method when passed the guessed letters string `"TOSX"`. The call below would ultimately return the character `'K'` in uppercase, even though the user typed `'k'`. Remember to match this console output format **exactly**. (*User input appears bold and blue*).

```
Your guess? s
You already guessed that letter.
Your guess? what?
Type a single letter from A-Z.
Your guess? X
You already guessed that letter.
Your guess? k
```

Remember that you are not limited to having only the above methods to help you implement this functionality, so if you want more decomposition you are welcome to add more methods. But you must have at least the methods shown above, with exactly those definitions. Your `playOneGame` code must also call these methods to help it solve the overall task of playing a single game.

Task 2: Display Hangman (ASCII Art)

The next feature to add is the display ("ASCII ART") of the hanging man on each turn. Since we know you got enough practice generating ASCII Art on Assignment 2, in the starter project we have provided several text files in the **res/** directory named **display0.txt** through **display8.txt**. These files contain the text that you should display each turn when the player has 0 guesses remaining through 8 guesses remaining, respectively. Here are the contents of some of these files:

display8.txt:

```
   +------------+
   |            |
   |
   |
   |
   |
   |
   |
   |
   |
   |
---+---
#######
```

display7.txt:

```
   +------------+          _________
   |            |         /         \
   |         _______      | Why me? |
   |        / o   o \   / \_________/
   |        |   .   |  /
   |        |  ___  |
   |        \_______/
   |            |
   |
   |
   |
---+---
#######
```

display6.txt:

```
   +------------+          _____________
   |            |         /             \
   |         _______      | I'm scared! |
   |        / o   o \   / \_____________/
   |        |   .   |  /
   |        |  ___  |
   |        \_______/
   |            |
   |            |
   |            |
   |            |
   |
   |
   |
   |
---+---
#######
```

... display0.txt:

```
   +------------+          _________________
   |            |         /                 \
   |         _______      | AUGH, I'M DEAD. |
   |        / X   X \   / \_________________/
   |        |   .   |  /
   |        |  ___  |
   |        \_______/
   |            |
   |      \\----+----//
   |            |
   |            |
   |           / \
   |          /   \
   |       __/     \__
   |
---+---
#######
```

`private void` **`displayHangman`**`(int guessCount)`

In this method you should print a drawing of the current Hangman state for the game based on the given number of guesses remaining. This amounts to reading and printing the contents of the file whose name corresponds to that number of guesses. For example, if the guess count is 3, your code should read and print the entire contents of the file ***res/display3.txt*** to the console. You may assume that the guess count is a legal value from 0-8 and that the given file exists in the res folder and can be read successfully. (It does not matter what code you put in your catch block, though we recommend a descriptive error message with the exception variable.)

You should also modify your `playOneGame` method so that on each turn, your program calls `displayHangman` to print the current game state, right before displaying the secret word hint, number of guesses remaining, and other game state. See the expected output logs in the `output/` folder of the starter project, or on the Assignment 3 webpage.

Secondary "canvas" console: The provided starter files come with a split-screen with a second console to the right of the main console. You can optionally choose to print your Hangman display to this secondary console instead of the main console while having the rest of the game output in the primary console, so that it is always visible on the screen during the game. To do so, use the command `canvas.println` rather than just `println`. You can also clear the canvas just before printing each file's contents.

```
canvas.clear();       // removes any text from the secondary console
canvas.println(text);// prints to the secondary console
...
```

Task 3: Choosing Random Words From a File

As you'll soon realize, the game is boring if the secret word is always `"PROGRAMMER"`; so now, you'll write a method that reads randomly-chosen new secret words for every game. You will do this by reading them from **input files** provided with the starter code. The starter code contains several dictionary data files to test with, such as **small.txt**, **medium.txt** and **large.txt**.

File format: Each input file uses exactly the following format (shown at right): the first line contains a single positive integer **n** that is the number of words in the file. Each of the following **n** lines contain a single uppercase word. You will read the input file for this program in the following method:

dict.txt:

```
73
ABSTRACT
AMBASSADOR
... (70 lines omitted)
ZIRCON
```

```
private String getRandomWord(String filename)
```

In this method you should read the file with the given name, and randomly choose and return a word from it. For example, if the filename passed is `"dict.txt"`, a provided dictionary file that contains 73 words, you should randomly choose one of those words and return it; each word should have equal probability of being chosen, (1/73 in this case).

Later in our course you will learn how to store large numbers of strings using features called *arrays* and *lists*. But we haven't learned such things yet, and you are not to use them on this assignment, even if you have somehow seen them before. The idea is not to read all of the words and store all of them as data in your program; instead, you must advance a Scanner to exactly the right place and select the word found there to be returned.

You may assume that the given file exists, can be read, and contains at least one word. (It does not matter what code you put in your catch block, though we recommend a descriptive error message with the exception variable.) You may also assume that the words in the given file are all uppercase, and that the integer count on the first line is correct.

Common bug: The **Scanner** has unusual behavior if you use `nextInt` and `nextLine` together. We recommend using `next` instead of `nextLine` to avoid such behavior, which works fine because each line contains only a single word.

Once you've written the above method, **modify your `run` method** to prompt the user for the dictionary filename. Do this just after printing the introduction. Your `run` method should use your new `getRandomWord` method to pull a random word from this file every time for each new game of Hangman. This way, each round will have a new random word.

The task of prompting and re-prompting for the file name can be made simple by using the `ConsoleProgram`'s method `promptUserForFile`, which accepts prompt string and directory parameters, and re-prompts until the user has typed the name of a file that exists in that directory. The function returns the file name that was typed. Note that the files in this project live in the `"res"` directory.

```
// re-prompt for valid filename in the given directory
String filename = promptUserForFile("prompt", "directory");
```

Task 4: Multiple Games and Statistics

Hopefully at this point your program nicely plays a single game of Hangman with a lovely ASCII display. Next, modify your `run` method so that your program is capable of playing multiple games. We aren't writing a new method for this part, just modifying `run` to have the new necessary code.

A key concept in this phase is that you should not need to go back to modify your methods from Tasks 1 or 2 (other than possibly fixing bugs). The new code you're adding here does not change the task of playing a single game or displaying the ASCII Hangman art. That's part of why we got those parts working first, so that we can build on them in this phase.

For this phase, when a game ends, prompt the user to play again. Do this from `run`, not from within `playOneGame`. If they type a `"Y"` or `"y"` response, play the game again. If they type a `"N"` or `"n"` response, end the program. At right is a partial log of the relevant part of the program; see the `output/` directory or the Assignment 3 webpage for complete output logs.

```
You win! My word was "PROGRAMMER".
Play again (Y/N)? ok
Illegal boolean format
Play again (Y/N)? Where am I?
Illegal boolean format
Play again (Y/N)? y
...
```

Your code should be robust against invalid input. Specifically, you should re-prompt the user until they type a Y or N, case-insensitively. `readBoolean` may come in handy here; as a reminder, it accepts parameters for the prompt message text, the "yes" text to look for, and the "no" text to look for (it ignores case), and it returns `true` or `false` to indicate yes or no. This means that you can use it as a test in an if statement or a while loop:

```
if (readBoolean("prompt text", "Y", "N")) {...
// or
while (readBoolean("prompt text", "Y", "N")) {...
```

Next, write code to output **statistics** about the player's games. Write the following method:

```
private void stats(int gamesCount, int gamesWon, int best)
```

In this method, you should print the final statistics that display after all games of Hangman have been played. You should print the total games played, number of games won, percentage of games won, and the game that required the fewest guesses to complete. (Of course, all of this information is passed to your method, so this method is fairly straightforward to write. The hard part is to keep track of the above statistics in other parts of your program so that you can pass proper values to stats later.) Your code should work for any number of games ≥ 1.

The following is the output from a call to `stats(4, 2, 5);` match our console output format **exactly**.

```
Overall statistics:
Games played: 4
Games won: 2
Win percent: 50.0%
Best game: 5 guess(es) remaining
Thanks for playing!
```

The stats are for a single run of the program only; if the user closes the program and runs it again, the stats are forgotten. If the player did not win any games, their win percentage is 0.0% and their best game is 0 guess(es) remaining.

Also modify your `run` method to call your new `stats` method once the user is done playing Hangman. At right is a partial log showing the relevant part of the program.

```
Play again (Y/N)? n
Overall statistics:
...
```

The hard part of this stage comes from figuring out how to pass around the data between your methods as parameters and return values. As a reminder, on this program you are forbidden from using private instance variables (aka "global" variables) so all data needed by your program must be stored in local variables and passed around using parameters and return so that run will have all of the necessary values to pass to `stats`. It can be challenging to find a good decomposition of the problem and pass the data around properly.

Optional Extra Features

There are many possibilities for extra features that you can add if you like for a small amount of extra credit. If you are going to do extra features, submit a **separate file** (e.g. "`HangmanExtra.java`") in your project containing your extended version; instructions on how to add a new file to an Eclipse project are on the Eclipse page of the course website, in the FAQ. In the comment header at the top of your `HangmanExtra.java` file, you must comment what extra features you completed. Here are some ideas for extra features:

- **Graphics:** The starter project includes a file `HangmanCanvas.java` in which you can write an optional graphical display of the Hangman game state. The existing Hangman canvas is used as a second console, as described previously; but you can replace this with your own code that draws shapes and colors to provide a more appealing display of the current game state. For example, the hanging man's head could be drawn as a `GOval` and his body as a `GLine`. As each guess is made, update your lines and shapes to represent the hanging man. The easiest way to do this is to call `removeAll();` on your canvas and start fresh.

  If you do the graphics extra feature, don't clutter your `HangmanExtra.java` with code to create shapes like `GOval`s and `GRect`s. Instead, have it call a **public** method that you write in the `HangmanCanvas` class, where you pass it the relevant information about the game (secret word, guess string, number of guesses left, etc.) as parameters, and the graphical class has the code to draw itself properly.

- **Sounds:** Make the game play a sound when a new game begins, when a correct or incorrect guess is made, when the game ends with a win or loss, and so on. The starter project does not contain any audio clip files, but you can find some on the web and download them into your project's **`res/`** subdirectory. You can load and play a sound in that directory named horn.wav by writing:

  ```
  AudioClip hornClip = MediaTools.loadAudioClip("res/horn.wav");
  hornClip.play();
  ```

  Note that for this you will need to add **`import acm.util.*`** and **`import java.applet.*`** to the top of your .java file.

- **Vary number of guesses allowed:** The default game lets the player have 8 guesses, but you could write a version that allows any number of guesses and prompts the user for how many they want to be given. Of course, you must figure out a way to make the hanging man's full body appear in that number of guesses.

- **Similar Games:** There are other games that involve guessing letters. Expand the program to play something like Wheel of Fortune, in which the word is now a phrase and in which you have to "buy" a vowel.

- **Other:** Use your imagination!

Grading

Functionality: Your code should compile without any errors or warnings. In grading we will test your game's overall flow, that it displays the right information at all times, handles user input properly, and so on. We use an Output Comparison Tool to see that your output exactly matches the output shown in this spec and the **`output/`** folder. You should use the Output Comparison Tool to verify your output before submitting.

Style: Follow style guidelines taught in class and listed in the course Style Guide. For example, use descriptive names for variables and methods. Format your code using indentation and whitespace. Avoid redundancy using methods, loops, and factoring. Use descriptive comments, including the top of each .java file, atop each method, inline on complex sections of code, and a citation of all sources you used to help write your program. If you complete any extra features, list them in your comments to make sure the grader knows what you completed. Limit yourself to using Java syntax taught in lecture and the textbook parts we have read so far. In particular, you are **forbidden from using instance variables**; a substantial deduction will be given if you do so. If there are important fixed values used in your code, declare them as constants.

Procedural decomposition: A highly important point of emphasis for style on this assignment is your ability to break down the problem into methods, both to capture redundant code and also to organize the code structure. Each method should perform a single clear, coherent task. No one method should do too large a share of the overall work. You must follow the specified decomposition of the problem into methods if you want to receive full credit for style. Do not change any of the required methods' names, parameters, or return values. Some students try to add extra parameters to these functions; you do not need any such extra parameters and will be heavily penalized for trying to do so. A big part of your style grade is using parameters and return values properly to send data between methods. A method can only return one value; if you want to return multiple values, consider using multiple smaller methods.

Your `run` method should represent a concise summary of the overall program, calling other methods to do much of the work of solving the problem. In particular, **`run` should not contain `println` or `print` statements**, though it can call other methods that print output.

Recursion: You should not write any method that **calls itself**. For example, don't have `playOneGame` call `playOneGame(...);` as a way of playing another game. This idea is called "recursion" and is not suitable for this assignment. Such a style is often desired by students who have trouble understanding returns. If you find yourself wanting to do that, find a different strategy involving a loop somewhere else in your code.

Honor Code: Follow the Honor Code when working on this assignment. Submit your own work and do not look at others' solutions (outside of your pair, if you are part of a pair). Do not give out your solution. Do not search online for solutions. Do not place a solution to this assignment on a public web site or forum. Solutions from this quarter, past quarters, and any solutions found online, will be electronically compared. If you need help on the assignment, please feel free to ask.