Stanford University, CS 106A, Homework Assignment 3
Hangman (PAIR ASSIGNMENT)

Based on past versions of this assignment created by Eric Roberts, Mehran Sahami, and others.

The purpose of this assignment is to create the game of Hangman to practice strings, file processing, parameters and returns. There is a **starter project ZIP archive** on the class web site. You will **turn in**:

- **Hangman.java**, your game implementation

This is a **pair assignment**. You may work in a pair, or you may work individually. If you're still **looking for a partner**, 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. Submit using the Submit Project entry in the Stanford Menu inside Eclipse.

The Game of Hangman:

(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 on course web site for complete output)

The program begins by printing an introductory message explaining the game to the player, followed by a series of games. The **output logs** on the web site shows sample executions of your program.

In each game, 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. The goal is to correctly guess all the letters in the secret word without making too many incorrect guesses.

Incorrect guesses are drawn 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-"`. If the player's guess does not appear in the secret word, the player is charged with an incorrect guess and a body part is added to the onscreen display of the hanging man.

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 he/she 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).

Your program should be **case-insensitive**; it should accept uppercase or lowercase letters and treat them the same.

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 for you to follow.
We also provide a list of specific **methods** that you are required to implement in your program.

<u>You are forbidden from using fields (private instance variables) on this assignment.</u>

Required Methods:

In past assignments we asked you to break apart the overall task into methods. You will do that again on this assignment, but since we are now using parameters and returns and writing a much larger program, it is difficult for a new programmer to come up with a good decomposition. Therefore we are going to tell you what headings the most important methods should have.

To help you structure your code properly and to make each piece of the program more testable, we **require** you to have at least the following methods. You can create additional methods if you like, but you must have these methods shown with **exactly** these names and exactly these parameters (no more, no less) and return types. Changing any of the headings of any of these methods will result in a large deduction; **do not change the headings** in any way.

In this section we will simply list the methods' names and headings; throughout this document we will describe the behavior each method should have.

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

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

A large part of the goal of this assignment is to practice using **parameters and returns** to communicate between these methods. This is why we are insisting on these specific methods and why we do not want you to modify their headings, parameters, or return types. Your program will not compile if it is missing any of the above methods.

You are not limited to having only the above methods, 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 headings. Your `playOneGame` code must also call the methods above to help it solve the overall task of playing a game.

You are forbidden from using fields (private instance variables) on this assignment.

Task 0: Introduction Message

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

```
public 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: Single Game

Start your development of the program by writing the code to play just a single game. Write the following method:

```
public 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 the given string as 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 multiple games or statistics. You will need code to do tasks such as the following:

- display a **"hint"** about the secret word; initially `"----------"`
- ask the user to type **guesses** (you can assume that the user types a valid guess each time, for now)
- figure out whether each guess is **correct** (found in the secret word) or incorrect (not in the secret word)
- keep track of the **number of guesses** remaining
- determine when the game has **ended** (when the user guesses all letters or runs out of guesses)

At right is a sample log of execution of this stage, using "PROGRAMMER" as secret word.
(User input appears bold and blue here for clarity.)

```
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".
```

You can test your program for now by changing the secret word passed by the `run` method.

You will need to keep a variable in your program that we will call the **"guessed letters"** string, which keeps track of every letter that the player has already guessed, both correctly and incorrectly. Start it empty but accumulate letters in it over time. For each unique guess, add it to the guessed letters string. To answer the question of, "Has a given letter already been guessed?", check if it's contained in the guessed letters string. The guessed letters string is also useful 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 upper case, even though all letters in the secret word are written in upper case. The guessed letters string should be all-uppercase.

Note that in Stage 1 and other stages, you will encounter **fencepost** issues much like the ones you saw in earlier assignments. Don't forget the techniques we saw in class for properly and elegantly solving fencepost problems.

(continued on next page)

Method Decomposition for Task 1:

Though you are still developing your program, you should take the time to decompose the problem into **methods**. We will identify major tasks that should be represented as methods, like asking the player for various information, generating hints, and so on. You will need to use **parameters and returns** to communicate between these methods.

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

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

In this method you should create and return a hint to the user. 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 `println` any console output.

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 `"--------"`

`public 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 a `char` in upper case. 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 execution of one call of this method when passed the guessed letters string `"TOSX"`. The call below would return the character `'K'` in upper case, even though the user typed `'k'`. Match our console output format **exactly**. *(User input is bold and blue in this document for clarity.)*

```
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
```

Once again, you are not limited to having <u>only</u> the above methods, 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 headings. Your `playOneGame` code must also call the methods above to help it solve the overall task of playing a game.

Task 2: Display Hangman (ASCII Art)

The next feature you can add to your game is the display ("ASCII art") of the hanging man on each turn.

```
public 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 does exist on the disk in the **res** folder and can be read successfully.

In the starter ZIP provided, we have given you several text files named **display0.txt** through **display8.txt**. in your project's **res/** directory. 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 \   / \_________________/
   |        |   .   |  /
   |        |  ___  |
   |        \_______/
   |            |
   |      \\----+----//
   |            |
   |            |
   |           / \
   |          /   \
   |       __/     \__
   |
---+---
#######
```

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 on the class web site.

When you write Java code to read a file, use a `Scanner` and enclose the code in a `try/catch` statement. For this program, you may assume that the **display*.txt** files exist in your project's **res/** folder and that they will be readable by your code. You can do anything you want in the `catch` block (the case when the file does not exist), but we suggest printing an error message including the exception variable itself, in case there are any issues in your project.

Secondary "canvas" console: The provided starter files come with a split-screen with a second console canvas 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, 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 canvas
    canvas.println(text);    // prints to secondary "canvas" console
    ...
```

Stage 3: Choosing Random Words from a File

The game is dull if it always uses `"PROGRAMMER"` as the secret word, so now you'll write code to use different secret words in every game. Let's write a method that reads randomly chosen new words for every game.

You will choose random words by reading them from **input files** provided with the starter code. The starter ZIP contains several dictionary data files to test with, such as **small.txt**, **medium.txt**, and **large.txt**.

dict.txt:

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

File format: Each input file uses exactly the following format: the first line contains a single positive integer *n* representing the number of words in the file. Each of the following *n* lines of the file contains a single word written in uppercase. You may assume that all files follow this format.

You will read the file of input for this program in the following method:

```
public String getRandomWord(String filename)
```

In this method you should read the given file, and randomly choose and return a word from it. For example, if the file name passed represents `"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 the case of `dict.txt`. Use a `Scanner` for reading files, and use a `RandomGenerator` for choosing random numbers.

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 the `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.) You may also assume that the words in the given file are all in uppercase and therefore that you don't need to capitalize/lowercase them as you read them in from the file one by one. You may assume 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 function, **modify your run method** to prompt the user for the dictionary file name. Do this just after the introduction has printed. Your `run` code should use your new `getRandomWord` method pull a random word from this file every time a new game of Hangman is started. This way every round will have a new random word to guess.

```
Guess wisely to avoid the gallows!

Dictionary filename? notfound.txt
Unable to open that file. Try again.
Dictionary filename? dict.txt
...
```

The task of prompting and re-prompting for the file name can be made simple by using the `ConsoleProgram`'s method **`promptUserForFile`**, which accepts a prompt string parameter and a directory, 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 file name in the given directory
String filename = promptUserForFile("prompt", "directory");
File file = new File(filename);
...
```

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. Above at right is a partial log of the relevant part of the program; see the course web site 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. The Stanford `ConsoleProgram` includes a useful method named `readBoolean` that can be used for exactly this purpose. It accepts parameters for the prompt message text, the "yes" text to look for, and the "no" text to look for, and it returns a `boolean` logical result of `true` or `false` to indicate yes or no. This means that you can use it as a test in an `if` statement or `while` loop:

```
    if (readBoolean("prompt text", "Y", "N")) { statements; }
// or:
    while (readBoolean("prompt text", "Y", "N")) { statements; }
```

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

`public 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(3, 2, 5);` Match our console output format **exactly**. The integers line up horizontally; use spaces to line them up in your output. **Round** the win% to 2 digits using **`printf`**.

```
Overall statistics:
Games played: 3
Games won:    2
Win percent:  66.67%
Best game:    5 guess(es) remaining
Thanks for playing!
```

The stats are for a single run of the program only; if they close the program and run 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:
...
```

Printing each individual stat is not very difficult. The hard part of this stage comes from figuring out how to pass around the data between your methods as **parameters and returns**. On this program you are **forbidden from using fields** (private instance variables, aka "global" variables), so all data needed by your program must be stored in local variables and passed around as parameters and returns 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.

Extra Features: *(Extra features are* *__optional__* *and will earn you a small amount of extra credit if you complete them.)*

There are many possibilities for extra features that you can add if you like. If you are going to do extra features, **submit two versions** of the assignment: the basic **Hangman.java** that meets all the assignment requirements, and a **HangmanExtra.java** containing your extended version. At the top of your **HangmanExtra.java** file in your comment header, you must **comment** what extra features you completed. Here are ideas for extra features:

- *Graphics:* The starter ZIP 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 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 / incorrect guess is made, when the game ends with a win / 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();
  ```

- *New ASCII art:* The provided **display*.txt** files do the job, but you might enjoy making your own files and drawing your own version of the hanging man at the gallows. Turn in a version of the game with a set of text files that you have drawn yourself, one for each unique game state as with the display0-8 .txt.

- *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.

- *Multiple languages:* Find a file of words from another language, such as Spanish or French. Your "extra" Hangman game could prompt the user for a language before playing and use a dictionary for that language.

- *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. What other features could you imagine in a game like this?

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 logs on the web site. You should use the Output Comparison Tool on the class web site to verify your output before turning in.

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, next to each field, 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. In general, limit yourself to using Java syntax taught in lecture and textbook parts we have read so far.

Fields: As mentioned, you are forbidden from using **fields** (a.k.a. instance variables). A <u>substantial</u> deduction will be given if you use fields. If there are important fixed values used in your code, declare them as static final **constants**.

<u>Once again, you are forbidden from using fields (private instance variables) on this assignment.</u>

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 <u>must</u> 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.

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

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, rather than a self-call.

Parameters and return: A big reason that we forbid fields here is because we want you to practice parameters and returns. Using them properly to send data between methods is a big part of your style grade. A method can only return one value; if you want to return multiple values, consider using multiple smaller methods.

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 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.