Chris Piech
CS 106A

Handout #12
Feb 7, 2018

Assignment #4 — Hangman
Due: 11am on Friday, Febuary 16th

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

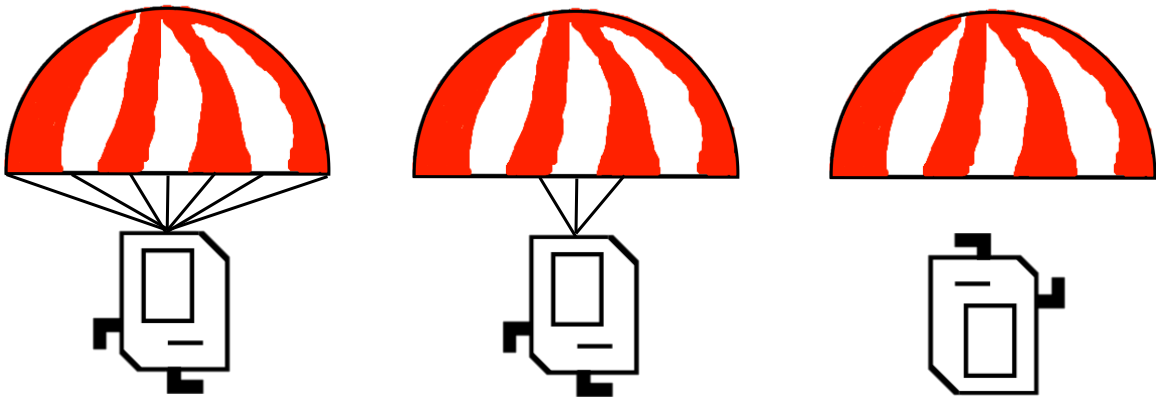
Y.E.A.H. hours Wednesday from 8:30-9:30pm in STLC 114.

Based on a handout by Eric Roberts

For this assignment, your mission is to write a program that plays the game of Hangman. The program is designed to give you some practice writing programs that manipulate strings and files.

When the user plays Hangman, the computer first selects a secret word at random from a list built into the program. The program then prints out a row of dashes—one for each letter in the secret word and asks the user to guess a letter. If the user guesses a letter that is in the word, the word is redisplayed with all instances of that letter shown in the correct positions, along with any letters correctly guessed on previous turns. If the letter does not appear in the word, the user is charged with an incorrect guess. The user keeps guessing letters until either (1) the user has correctly guessed all the letters in the word or (2) the user has made seven incorrect guesses. Two sample runs that illustrate the play of the game are shown in Figure 1 on the next page.

When it is played by children, the real fascination (a somewhat morbid fascination, I suppose) from Hangman comes from the fact that incorrect guesses are recorded by drawing an evolving picture of the user coming to their demise. In this version we are going to draw Karel attached to a parachute with a number of cords equal to the number of guesses left. At the start of the game, Karel has seven cords (pictured on the left). As you run low on guesses Karel runs low on cords (pictured in the middle). When you are out of guesses, Karel is out of cords (pictured on the right)… sad times.



To get you warmed up we ask that you first write a minimal program (sandcastle) that leverages the essential concepts needed for Hangman:

Sandcastle: Alternate Caps

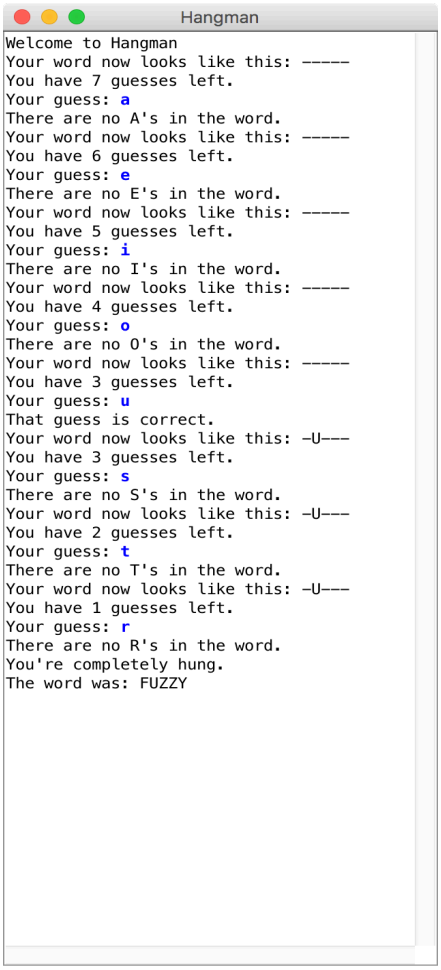
Write a method `altCaps(String input)` which converts a string to alternating capital letters, meaning you alternate between uppercase and lowercase. This style of typing was prevalent on the internet in the late 90s. For example:

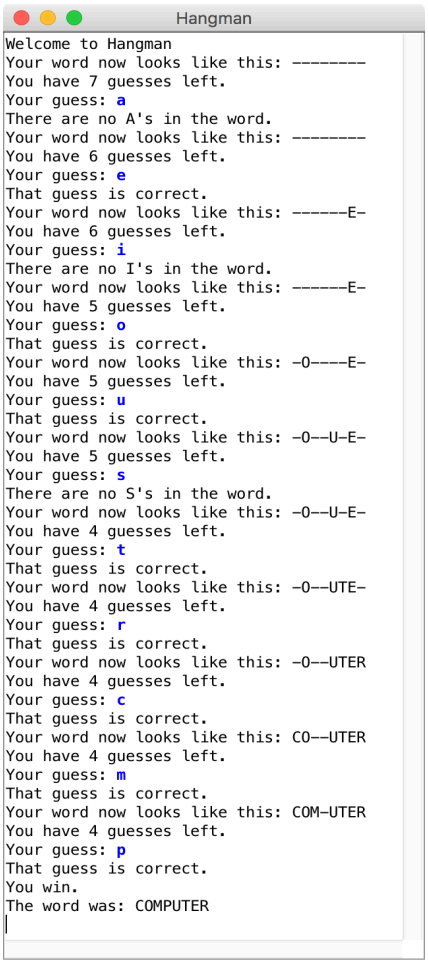
```
altCaps("aaaaaa")          returns   "aAaAaA"
altCaps("hello world")     returns   "hElLo WoRlD"
```

Note that characters that are not letters are not changed and do not affect the alternating sequence of uppercase and lowercase letters.

Design and test your Hangman program in three parts. The first part gets the text game working without any graphics at all and with a fixed set of secret words. The second part consists of drawing the graphics for Hangman. The final part requires you to replace the supplied version of the secret word list with one that reads words from a file.

Figure 1. Two sample runs of the Hangman program (console only)





Note that your program only needs to be able to play the Hangman game once through (i.e., the player guessing one word), but it should be pretty easy to extend your program to allow the player to play multiple rounds.

Part I—Playing a console-based game

In the first part of this assignment, your job is to write a program that handles the user interaction component of the game—everything except the graphical display. To solve the problem, your program must be able to:

- Choose a random word to use as the secret word. That word is chosen from a word list, as described in the following paragraph.
- Keep track of the user's partially guessed word, which begins as a series of dashes and then gets updated as correct letters are guessed.
- Implement the basic control structure and manage the details (ask the user to guess a letter, keep track of the number of guesses remaining, print out the various messages, detect the end of the game, and so forth).

The only operation that is beyond your current knowledge is that of representing the list of words from which you can choose a word at random. For the first two parts of the assignment, you will simply make use of a method that we've given you called **`getRandomWord`**. The implementation of the method you've been given is only a temporary expedient to make it possible to code the rest of the assignment. In Part III, you will replace the definition we've provided with one that utilizes a list of words from a data file.

The strategy of creating a temporary implementation that provides enough functionality to implement the rest of the program is a common technique in programming. Such temporary implementations are usually called **stubs.** A game that used this implementation of **`getRandomWord`** would quickly become uninteresting because there are only ten words available. Even so, it will allow you to develop the rest of the program and then come back and improve this part later.

Part I is a string manipulation problem using the methods developed in Chapter 8. The sample runs in Figure 1 should be sufficient to illustrate the basic operation of the game, but the following points may help to clarify a few issues:

- You should accept the user's guesses in **either lower or upper case**, even though all letters in the secret words are written in upper case.
- If the user guesses something other than a single letter, your program should tell the user that the guess is illegal and accept a new guess.
- If the user guesses a correct letter more than once, your program should simply do nothing. Guessing an incorrect letter a second time should be counted as another wrong guess. (In each case, these interpretations are the easiest way to handle the situation, and your program will probably do the right thing even if you don't think about these cases in detail.).

Figure 2. Stub implementation of HangmanLexicon

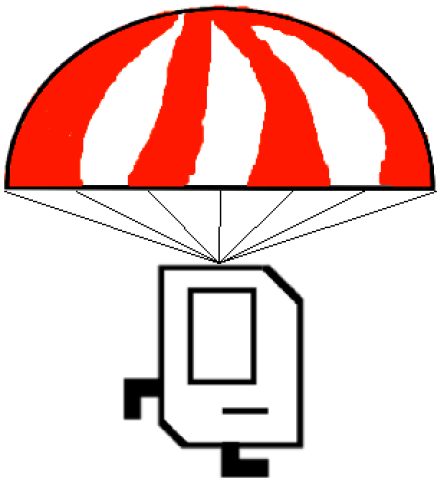
```
/**
 * Method: Get Random Word
 * -----------------------
 * This method returns a word to use in the hangman game. It randomly
 * selects from among 10 choices.
 */

private String getRandomWord() {
   int index = rg.nextInt(10);
   if(index == 0) return "BUOY";
   if(index == 1) return "COMPUTER";
   if(index == 2) return "CONNOISSEUR";
   if(index == 3) return "DEHYDRATE";
   if(index == 4) return "FUZZY";
   if(index == 5) return "HUBBUB";
   if(index == 6) return "KEYHOLE";
   if(index == 7) return "QUAGMIRE";
   if(index == 8) return "SLITHER";
   if(index == 9) return "ZIRCON";
   throw new ErrorException("getWord: Illegal index");
}
```

Remember to finish Part I before moving on to Part II. Part II is arguably more fun, but it is essential to develop large programs in manageable stages.

Part II—Adding graphics

For Part II, your task is simply to extend the program you have already written so that it now keeps track of the Hangman graphical display. Although you might want to spice things up in your extensions, the simple version of the Karel parachute looks like this:



The first thing you should do when you begin Part II is to create a new **GCanvas** and install it in the program window next to the console. The **Hangman** class itself is an instance of a **ConsoleProgram**, which means that the startup code in the ACM libraries has installed an **IOConsole** in the window that fills the entire space. Your next task is to add a **GCanvas** to the program window as well. The code you need for this part is extremely simple. First, in the instance variables section of the **Hangman** program, you need to declare an instance variable for the canvas by writing

```
private GCanvas canvas = new GCanvas();
```

and then add the following **init** method to your program:

```
public void init() {
   add(canvas);
}
```

Note that your **Hangman** program will have both an **init** and a **run** method as a result, and that is perfectly fine. **init** is a method that gets executed before the program window is displayed. This **init** method adds the canvas to the window prior to the **run** method being executed; the **run** method is where the execution of your game will start after the window is initialized. By default, the contents of the program window are given equal amounts of space side by side. Since this is a console program, the console is already installed and will therefore show up in the left column. When you add the **GCanvas** it will occupy the second column, which means that the console and graphics components of the window will each get half the screen area, as shown in Figure 4 below. Input and output from the **Hangman** program will continue to appear on the console, and any objects you add to the variable **canvas** will appear in the area on the right.

Figure 3. Screen shot showing side-by-side console (left) and canvas (right)

Importantly: All **GraphicsProgram** methods now need to be called on the canvas object. This is because a **ConsoleProgram** does not know how to handle graphics logic.

When you want to add **GObject**s to the screen, you need to use **canvas.add(object)**. Instead of using **getWidth()** or **removeAll()**, use **canvas.getWidth()** or **canvas.removeAll()**.

This method demonstrates how to draw an image in a console program with an added **GCanvas** called **canvas**:

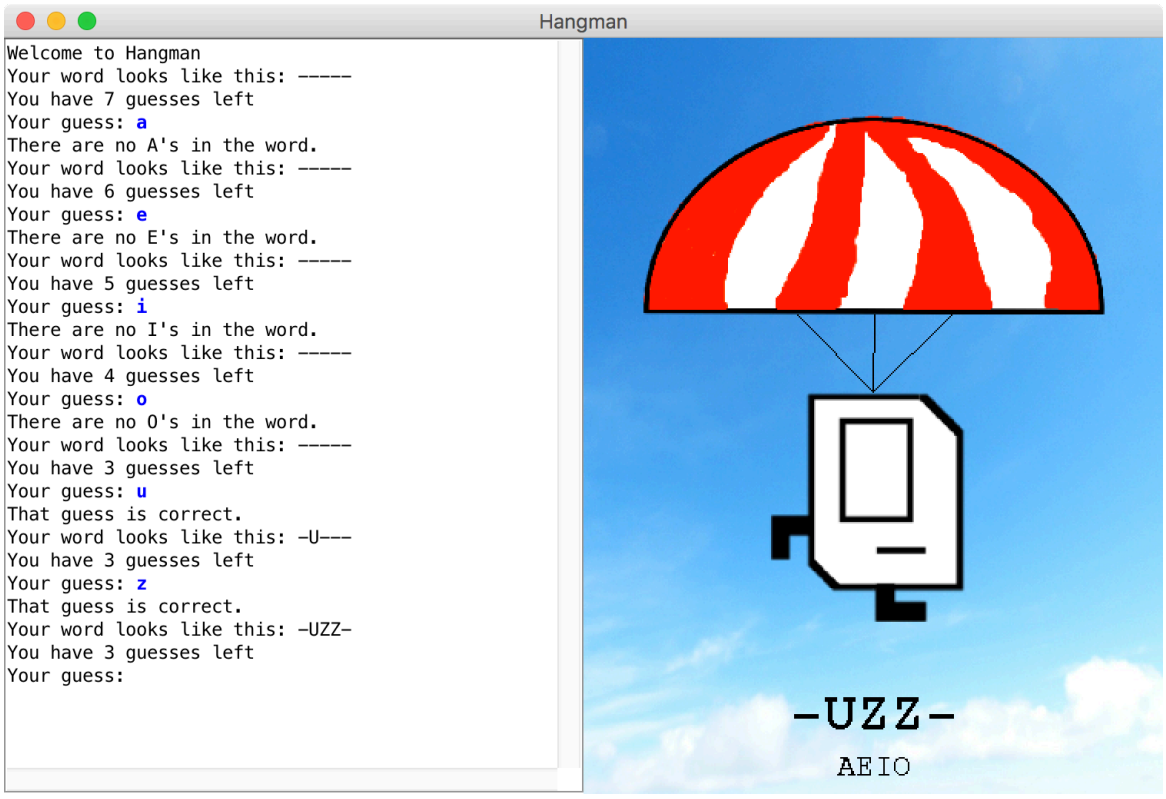
```
private void drawBackground() {
   GImage bg = new GImage("background.jpg");
   bg.setSize(canvas.getWidth(), canvas.getHeight());
   canvas.add(bg, 0, 0);
}
```

At all times in the program you should display: a background, Karel with her parachute, the partially guessed word and the user's incorrect guesses. The graphics use four different images:

File Name	Description
"background.jpg"	has the nice sky background,
"karel.png"	has the Karel image.
"parachute.png"	has the parachute image.
"karelFlipped.png"	has a picture of Karel upside down.

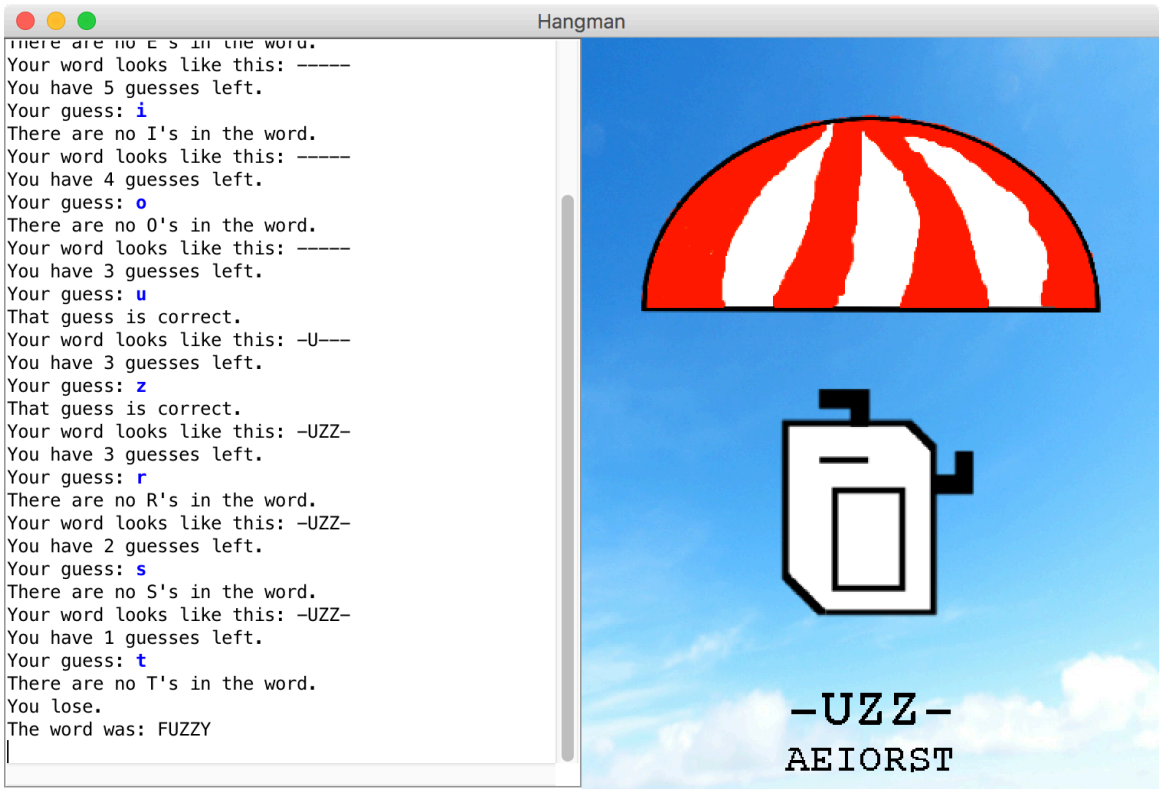
The size and y-location of the images and text (as well as the text fonts) are stored as constants. Make sure that all objects are centered. Karel should initially be connected to the parachute by seven lines which are evenly spaced along the bottom of the parachute,

Figure 4. Midway through a graphical hangman game



and connect to the top-center of Karel. As the user guesses letters incorrectly the cords should break from outside in. In other words first break the furthest most right string, then break the furthest most left string and so forth. Figure 4 the end of a session in which the user is trying to guess **FUZZY**. When Karel is out of strings, you should show her upside down to show that she is free falling as in Figure 5. Sorry Karel.

Figure 5. When you run out of guesses, Karel runs out of hope



Part III—Reading the lexicon from a data file

Your job in this part of the assignment is simply to re-implement the **getRandomWord** method so that instead of selecting from a meager list of ten words, it reads a much larger word list from a file. The steps involved in this part of the assignment are as follows:

1. Open the data file **HangmanLexicon.txt** using a **BufferedReader** that will allow you to read it line by line.
2. Read the lines from the file into an **ArrayList**.
3. Reimplement the **getRandomWord** method so that it uses the **ArrayList** from step 2 as the source of the words.

The first two steps should be at the start of your program.

Note that methods which use **getRandomWord** should not have to change in response to this change in the implementation. Insulating parts of a program from changes in other parts is a fundamental principle of good software design.

Extensions

There are many ways to extend Hangman to make it more fun. Here are some ideas:

- You could spice up the display a little. The static image of Karel seems a bit tame here.
- You could animate the pictures. Be creative!
- Allow the user to play multiple games.
- Once you get the basic structure working, you could expand the program to play something like Wheel of Fortune, in which the single word is replaced by a common phrase and in which you have to buy vowels
- You could write an A.I. agent that plays the game for the user. One common strategy is to guess the most frequent letter. After loading words from the lexicon, find out which letters come up most often.
- Use your imagination!