CS 106A Section 8 Handout (Week 9)

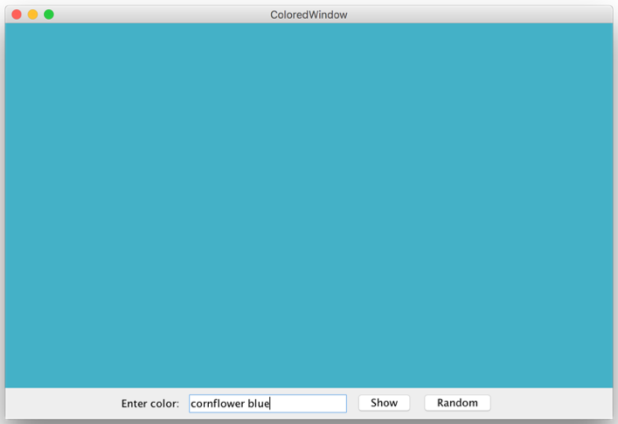
Overview: these problems will give you practice with writing programs with *interactors* such as `JButtons`, `JTextFields` and `JCheckBoxes`. You will see two approaches to writing interactor programs: extending `Program` and extending `GraphicsProgram`. In both cases we add our interactors in the *init()* method. The difference is that you must extend `Program` if you want to use your own custom subclass of `GCanvas` instead of the one that `GraphicsProgram` creates for you. This lets you better organize your code since you can put graphics code in your canvas and interactor code in your main program file. This approach is **required** in NameSurfer; however, it's not necessary in shorter programs such as the first problem below where we have little canvas-related code.

1. **ColoredWindow**. Many years back, the website *xkcd* ran an experiment where it asked users to name a variety of colors. Over 5 million colors and 140,000 participants later, they aggregated a collection of various colors with their (sometimes eclectic) labeled names, from *khaki* to *baby blue* to *sandy brown* to *radioactive green* (full list at https://xkcd.com/color/rgb/). Your job is to write a `GraphicsProgram` that allows the user to type in one of these names to view its corresponding color. The user can press "Show" or press ENTER to view the color. If the name the user entered is not included in the data, your program should do nothing. There should also be a "Random" button that picks a random color name and displays its name in the text field and the color onscreen.

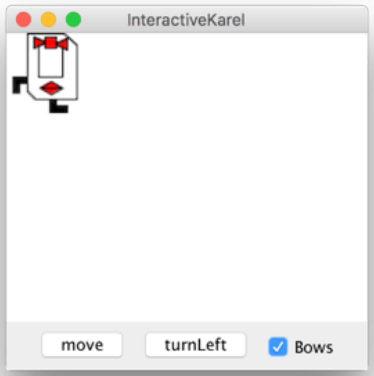
   You should get the red/green/blue values for the named colors from the experiment from the given text data file, **`res/colors.txt`**. This file contains the name of a color on one line and the values of the red/green/blue components for that color on the next line. To create a `Color` out of its red, green, and blue components, you can use the `Color` constructor by writing `new Color(r, g, b)`. To avoid reading the file every time the user wants to view a new color, store the colors in a **data structure**.

   Example colors.txt

```
pastel blue
72 100 175
baby blue
182 226 245
purple
130 64 234
blue
75 49 234
olive green
111 145 122
...and so on...
```

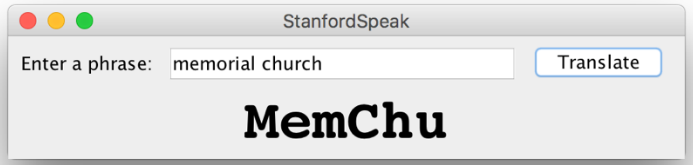


2. **DIY Karel.** Write a Java program that imitates some aspects of how Karel the Robot behaves. As shown in the diagram to the right, you should create a **move** button and a **turnLeft** button, along with a check box to optionally display Karel with bows. Pressing move causes Karel to move one length in the direction it is facing. If moving Karel would cause it to move off the screen, just keep Karel where it is. Pressing turnLeft causes Karel to turn left. Toggling the checkbox alternates showing Karel with and without bows. You can assume you are given four images for each of the directions that Karel can face, and for Karel with and without bows, called `KarelEast.jpg`, `KarelWest.jpg`, `KarelSouth.jpg`, `KarelNorth.jpg`, `BowKarelEast.jpg`, `BowKarelWest.jpg`, etc. that are 64 pixels in width and height.

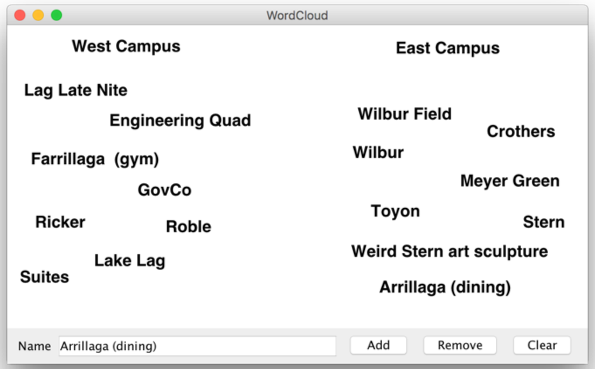


3. **Stanford Speak.** Stanford students have abbreviations for pretty much everything, from *CoHo* for Coffee House to *MemChu* for Memorial Church. These abbreviations are formed by combining the initial substrings of each word into a single string. Write a program like the one below that helps incoming freshmen learn "Stanford Speak" by translating English phrases into their Stanford abbreviations. The user should be able to click "Translate" or press ENTER to view a translation.

   You may assume that you have a method `abbreviateWord` that, given a single word, returns its abbreviated form. For example, calling `abbreviateWord("memorial")` returns `"Mem"`.



4. **Word Cloud.** "Word Clouds" are graphical arrangements of words into different clusters. They are often used to organize ideas, show similarities, and create other interesting visualizations. For this problem, write a program that lets a user design their own word cloud:



The program should have the following interactors: a "Name" label, a text field, and buttons to add, remove, and clear labels. To **add** a label to the screen, the user should be able to type text into the text field and click "Add" to add it. Labels should initially be added at the center of the screen. To **remove** a label from the screen, the user should be able to type in the text of the onscreen label they want to remove and click "Remove". You may assume that each label's text is unique. To **clear** all labels from the screen, the user should be able to click "Clear".

Once a label has been added to the screen, the user should be able to click and drag the label around with the mouse to reposition it.

Note: if the only objects in the window were labels, you could implement the clear button by removing everything from the canvas. While that would work in this case, you might want to extend the program so if there were other objects on the screen that were part of the application, they would remain onscreen. In that case, you would want to implement clear by iterating through all onscreen labels and removing each one.