Nick Troccoli
CS 106A

Section #7
August 9, 2017

Section Handout #7: Interactors and Data Structures

Parts of this handout by Mehran Sahami, Eric Roberts, Marty Stepp, and Patrick Young

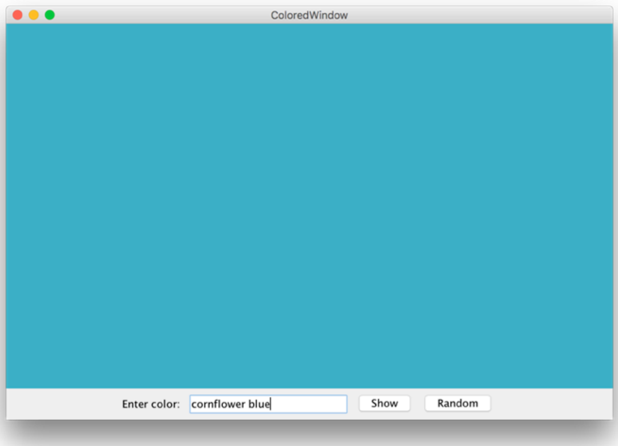
1. ColoredWindow.

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**.

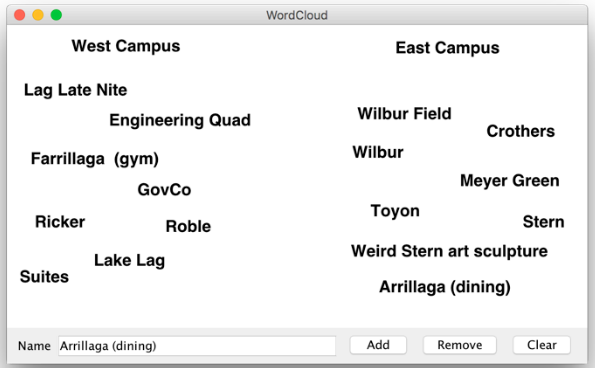
Example `colors.txt`

```
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. 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:



Name: Arrillaga (dining)

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”.

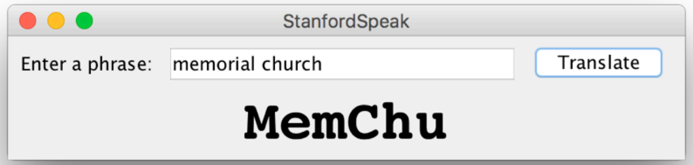
Once a label has been added to the screen, the user should be able to click and drag the label around with the mouse to reposition it.

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.

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



Enter a phrase: memorial church

MemChu

word, returns its abbreviated form. For example, calling `abbreviateWord("memorial")` returns "`Mem`".

4. Data structure design

So far in CS106A, we've worked a good deal with arrays and ArrayLists. While arrays have fixed sized, ArrayLists grow as more elements are added (usually, to the end). We could think of potentially an even more powerful idea: an expandable array. The idea here is that we could think of an array that dynamically expanded to accomodate whatever index we try to access. For example, if we started with an empty array, and tried to add an element at index 14, the array would automatically grow large enough, so that elements 0 through 14 all existed (and we could store the given value at index 14). All the elements of the array that had not previously been given a value would have the value **`null`**. Then if we tried store a value at, say, index 21, the array would again grow automatically to have space for elements up to and including index 21. Note that the value at index 14 would still appear to be at index 14 in the expanded array.

Being able to expand an array dynamically is useful enough that it might be worth creating an abstraction to implement it. Such an abstraction is shown in Figure 1 (on the next page), which shows the structure of an **`ExpandableArray`** class that allows the client to call **`set`** on an index even if it doesn't currently exist in the array. When such a situation occurs, the implementation of the **`ExpandableArray`** class has to allocate a new internal array that is large enough to hold the desired element and then copy all of the existing elements into the new array. The point of this problem is simply to make that operation general by creating a new class that provides the functionality, and we make the class sufficiently general by having it be able to store any type of object (hence the use of the type **`Object`** for the array elements). To create an expandable array, and set some of its values (which in this case are Strings), you might have code such as:

```
ExpandableArray myList = new ExpandableArray();
myList.set(14, "Bob");
myList.set(21, "Sally");
```

When you wanted to retrieve the value at a particular element of the expandable array, you could do something like the following:

```
String value = (String) myList.get(14);  // Note the cast
if (value != null) {
   println("Got value: " + value);
}
```

In writing your solution, you should keep the following points in mind:

- The underlying structure in your implementation must be an array containing elements of the type **`Object`**. Although using a **`HashMap`** might well be easier, it will be less efficient than the array in terms of the time necessary to look up a specific index.

- Notice that the definition of the abstraction explicitly states that the **`get`** method should return **`null`** if the specified index value does not exist. That means that you will need to check whether the index is out of bounds before trying to select that element.
- You may assume that all of the index values supplied by the client are nonnegative and do not need to check for such values in the implementations of **`get`** and **`set`**.

Figure 1. Proposed structure of the ExpandableArray class

```
/**
 * This class provides methods for working with an array that expands
 * to include any positive index value supplied by the caller.
 */

public class ExpandableArray {

/**
 * Creates a new expandable array with no elements.
 */
   public ExpandableArray() {
      . . . You fill in the implementation . . .
   }


/**
 * Sets the element at the given index position to the specified.
 * value. If the internal array is not large enough to contain that
 * element, the implementation expands the array to make room.
 */
   public void set(int index, Object value) {
      . . . You fill in the implementation . . .
   }


/**
 * Returns the element at the specified index position, or null if
 * no such element exists.  Note that this method never throws an
 * out-of-bounds exception; if the index is outside the bounds of
 * the array, the return value is simply null.
 */
   public Object get(int index) {
      . . . You fill in the implementation . . .
   }
}
```