Sarai Gould & Laura Cruz-Albrecht
CS 106A

Section #6
July 31, 2019

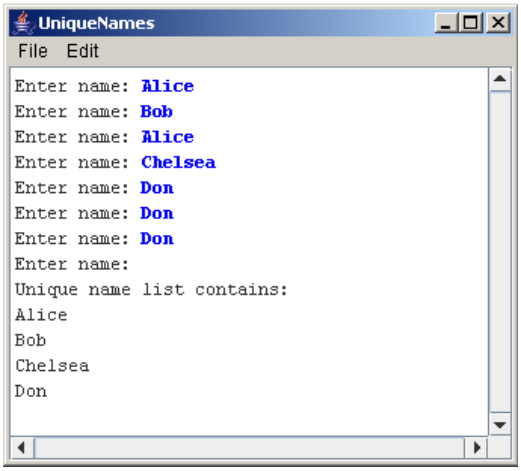
Section Handout #6: ArrayLists, HashMaps, and Classes

Portions of this handout by Eric Roberts, Marty Stepp, Chris Piech, Mehran Sahami, and Julia Daniel

ArrayLists

1. How Unique!

Write a program that asks the user for a list of names (one per line) until the user enters a blank line (i.e., just hits return when asked for a name). At that point the program should print out the list of names entered, where each name is listed only once (i.e., uniquely) no matter how many times the user entered the name in the program. You may find that using an **`ArrayList`** to keep track of the names entered by the user may greatly simplify this problem. A sample run of this program is shown at right.



2. Remove Even Length

Write a method named **`removeEvenLength`** that takes an ArrayList of strings as a parameter and removes all of the strings of even length from the list. For example, if an ArrayList variable named list contains the values `["hi", "there", "how", "is", "it", "going", "good", "folks"]`, the call of **`removeEvenLength`**`(list);` would change it to store `["there", "how", "going", "folks"]`.

3. Switch Pairs

Write a method **`switchPairs`** that switches the order of values in an **`ArrayList`** of strings in a pairwise fashion. Your method should switch the order of the first two values, then switch the order of the next two, switch the order of the next two, and so on. For example, if an `ArrayList` variable named list initially stores these values:

```
["four", "score", "and", "seven", "years", "ago"]
```

Your method should switch the first pair, "four" and "score", the second pair, "and" and "seven", and the third pair, "years", "ago". So the call of `switchPairs(list);` would yield this list:

```
["score", "four", "seven", "and", "ago", "years"]
```

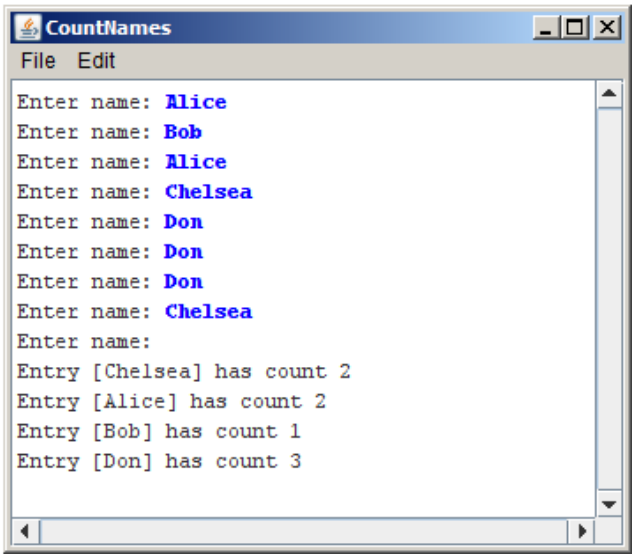
If there are an odd number of values, the final element should not be moved (such as "hamlet" below):

```
["to", "be", "or", "not", "to", "be", "hamlet"]
```

HashMaps

4. Name Counts

Write a program that asks the user for a list of names (one per line) until the user enters a blank line (i.e., just hits return when asked for a name). At that point the program should print out ***how many times*** each name in the list was entered. You may find that using a **`HashMap`** to keep track of the information entered by the user may greatly simplify this problem. A sample run of this program is shown below.



5. Mutual Friends

In the days before social networks, one of the easier ways to find mutual friends was to compare address books and find common entries. Write a method named **`mutualFriends`** that accepts as parameters two **`HashMaps`** from strings to integers representing two phonebooks, and returns a new map containing only the key/value pairs that exist in both of the phonebooks. Remember that for an entry to be included in your result map, not only do both maps need to contain that key, but they need to map that key to the **same value** (phone number). For example, consider the following two maps:

```
{Jennie=8675309, Jonah=2124320, Colin=4602121, Annie=4444444,
                        Avery=8080543}

{Matthew=6202121, Pooja=8888888, Jonah=2124320, Colin=4602121,
                Annie=4444444, Jennie=2128765}
```

Calling your method on the preceding maps would return the following new map (the order of the key/value pairs does not matter):

```
{Jonah=2124320, Colin=4602121, Annie=4444444}
```

Notice how even though the key `Jennie` is present in both maps, it is not included in our final result map because it maps to a different phone number in each phonebook.

6. Reverse

Write a method called **`reverse`** that accepts a `HashMap` from integers to strings as a parameter and returns a new `HashMap` of strings to integers that is the original's "reverse". The reverse of a map is defined here to be a new map that uses the values from the original as its keys and the keys from the original as its values. Since a map's values need not be unique but its keys must be, it is acceptable to have any of the original keys as the value in the result. In other words, if the original map has pairs (k1, v) and (k2, v), the new map must contain either the pair (v, k1) or (v, k2).

For example, for the following map:

```
{42=Marty, 81=Cynthia, 17=Dan, 31=Emma, 56=Dan, 3=Marty, 29=Dan}
```

Your method could return the following new map (the order of the key/value pairs does not matter):

```
{Marty=3, Cynthia=81, Dan=29, Emma=31}
```

Classes

7. Student

Define a class named **`Student`**. A **`Student`** object represents a Stanford student that, for simplicity, just has a name, ID number, and number of units earned towards graduation. Each **`Student`** object should have the following public behavior:

`new Student(`***`name, id`***`)`	Constructor that initializes a new Student object storing the given name and ID number. The unit count should initially be 0.
`s``.getName();` ***`s`***`.getID();` ***`s`***`.getUnits();`	Returns the name, ID, or unit count of the student, respectively.
`s``.incrementUnits(`***`units`***`);`	Adds the given number of units to this student's unit count.
`s``.hasEnoughUnits();`	Returns whether the student has enough units (180) to graduate.
`s``.toString();`	Returns the student's string representation, e.g. "Colin (#42342)".

8. Paper Plane Airport

Write two classes, an **`Airport`** class and an **`Airplane`** class, which work together to create and dispatch (paper) airplanes.

The **`Airport`** class should be a program that manages the manufacturing and dispatch of airplanes. It should be able to build **`Airplane`**s, keep track of **`Airplane`**s that have been built, and tell **`Airplane`**s to **`takeOff()`**. Write a program that builds 3 airplanes, dispatches 2 of them, builds one more, and then dispatches all airplanes that haven't been dispatched yet. *(hint: **`Airport`** should have a **`run()`** method)* Consider what scheme out of all the options we've learned so far might make sense for keeping track of airplanes – there is no one right answer, but some might be simpler or more generalizable than others for the purposes of this task.

The **`Airplane`** class should be a special variable type that can build (*aka construct*) a new **`Airplane`**, keep track of whether the **`Airplane`** is airborne, and implement how an **`Airplane`** can **`takeOff()`**. In order to actually build a paper airplane, you have to fold paper in half and then fold the wings out. You can assume that you already have two methods, **`foldInHalf()`** and **`foldWings()`**, which can be called with no parameters or return values in order to do so. You don't need to write the code for these two methods yourself – just call them – but do consider: should **`foldInHalf()`** and **`foldWings()`** be public or private methods in **`Airplane`**? How can a user find out whether an **`Airplane`** is airborne?

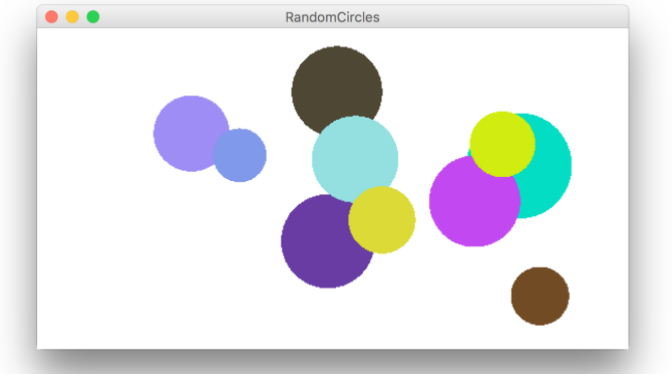
A note on scope and decomposition, as they relate to classes: the **`Airport`** class doesn't need to know how a single **`Airplane`** is built or its internal workings; all it cares about (and should be able to do) is making, monitoring, and dispatching **`Airplane`**s according to the instructions above. Similarly, the **`Airplane`** class doesn't need to know how many airplanes are built, what data structure(s) they may be stored in, or other information about how **`Airplane`**s are managed; all it cares about is the inner workings and status of a single **`Airplane`**. Maintaining this "wall of abstraction" is key to using classes properly!

9. Subclassing GCanvas

When defining your own classes, you can also **`extend`** classes that already exist. This essentially means that the class you are defining can inherit the behavior of the class it is extending, and can then build on top of it with additional behavior. One example of this is subclassing **`GCanvas`**. We can do this if we want to make our own canvas with additional behavior beyond a standard **`GCanvas`**. This also lets us put our graphics code in the **`GCanvas`** subclass file instead of inside our main program.

For this problem, write a **`GCanvas`** subclass **`RandomCirclesCanvas`** that implements similar behavior to the "Random Circles" problem from Section 3. As a quick refresher, that program drew **`N_CIRCLES`** random circles on the canvas, where each circle had a randomly chosen color, a randomly chosen radius between 5 and 50 pixels, and a randomly chosen position on the canvas, subject to the condition that the entire circle

must fit inside the canvas without extending past the edge. The following shows one possible sample run:

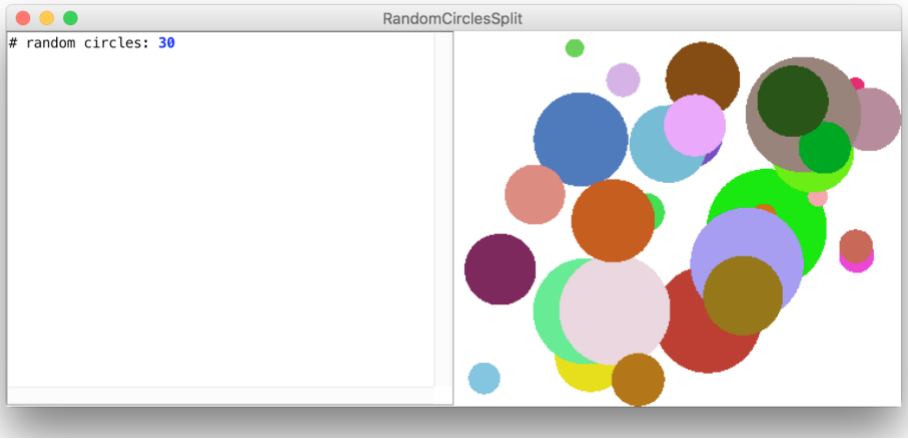


For this version, `RandomCirclesCanvas` should implement a method `drawRandomCircle` that draws a single random circle, subject to the constraints listed above. The main program file that uses this class is included below:

```
/*
 *  File: RandomCircles.java
 *  ------------------------
 *  This program draws a set of 10 circles with random sizes,
 *  positions, and colors.
 */

import acm.program.*;

public class RandomCircles extends Program {

	/** Number of circles */
	private static final int NCIRCLES = 10;
	RandomCirclesCanvas canvas;

	public void init() {
		canvas = new RandomCirclesCanvas();
		add(canvas);
	}

	public void run() {
		for (int i = 0; i < NCIRCLES; i++) {
			canvas.drawRandomCircle();
		}
	}
}
```

Extra: One nice stylistic note about defining a `GCanvas` sublass is that it's not tied to a specific Graphics or Console program, since it's in its own file. For instance, it's easy for another programmer to come along and make a variation of this program using your canvas, but in a *split-screen* program that prompts the user for the number of circles to

draw. The following sample run shows one possible outcome of this split-screen program:



The code for this modified version is as follows, again using the same `RandomCirclesCanvas` class we defined earlier:

```
/*
 * File: RandomCirclesSplit.java
 * -----------------------
 * This program draws a set of circles with random sizes,
 * positions, and colors. The number of circles drawn is
 * given by the user.
 */

import acm.program.*;

public class RandomCirclesSplit extends ConsoleProgram {
	RandomCirclesCanvas canvas;

	public void init() {
		canvas = new RandomCirclesCanvas();
		add(canvas);
	}

	public void run() {
		int numCircles = readInt("# random circles: ");
		for (int i = 0; i < numCircles; i++) {
			canvas.drawRandomCircle();
		}
	}
}
```