Nick Troccoli
CS 106A

Section #6
August 1, 2017

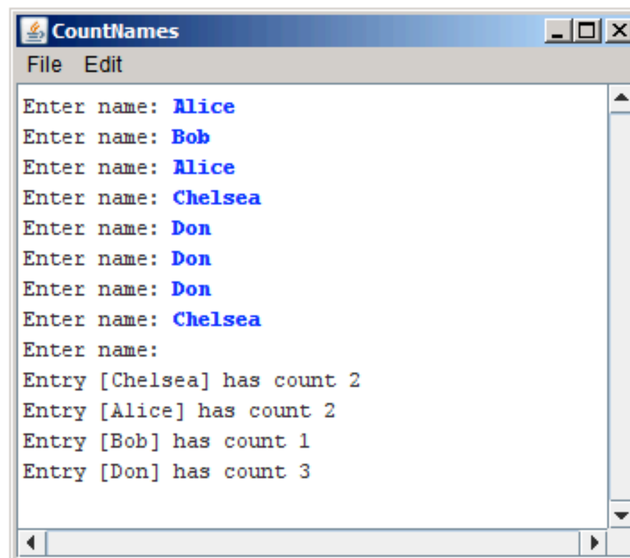
Section Handout #6: HashMaps, ArrayLists, and Classes

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

HashMaps

1. 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 user may greatly simplify this problem. A sample run of this program is shown below.



2. Intersect

Write a method called intersect that accepts two HashMaps from strings to integers as parameters and returns a new map containing only the key/value pairs that exist in both of the parameter maps. For a key/value pair to be included in your result, not only do both parameter maps need to contain a mapping for that key, but they need to map that key to the same value. For example, consider the following two maps:

```
{Janet=87, Logan=62, Whitaker=46, Alyssa=100, Stefanie=80,
                Jeff=88, Kim=52, Sylvia=95}

{Logan=62, Kim=52, Whitaker=52, Jeff=88, Stefanie=80, Brian=60,
                Lisa=83, Sylvia=87}
```

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

`{Logan=62, Stefanie=80, Jeff=88, Kim=52}`

3. 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=Sue, 17=Ed, 31=Dave, 56=Ed, 3=Marty, 29=Ed}`

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

`{Marty=3, Sue=81, Ed=29, Dave=31}`

<u>Array Lists</u>

4. 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 user may greatly simplify this problem.

A sample run of this program is shown below.

5. 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", "sirs"]`, the call of `removeEvenLength(list);` would change it to store `["there", "how", "going"]`.

6. Mirror.

Write a method named mirror that accepts an `ArrayList` of strings as a parameter and produces a mirrored copy of the list as output. For example, if an `ArrayList` variable

named list contains the values on the left before your method is called, after a call of mirror(list); it should contain the values on the right:

```
["how", "are", "you"] => ["how", "are", "you", "you", "are", "how"]
```

7. Switch Pairs.

Write a method named 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"]
```

Classes

8. 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. "Nick (#42342)".

9. TimeSpan. Define a class named TimeSpan. A TimeSpan object stores a span of time in hours and minutes (for example, the time span between 8:00am and 10:30am is 2 hours, 30 minutes). Each TimeSpan object should have the following public behavior:

`new TimeSpan(`***`hours`***`, `***`minutes`***`)`	Constructor that initializes a new TimeSpan object storing the given time span of minutes and seconds. Assume that the parameters are valid: the hours are non-negative, and the minutes are 0 - 59.
`ts``.add(`***`hours`***`, `***`minutes`***`);`	Adds the given amount of time to the span. For example, (2 hours, 15 min) + (1 hour, 45 min) = (4 hours). Assume that the parameters are valid: the hours are non-negative, and the minutes are between 0 and 59.
`ts``.getHours();` ***`ts`***`.getMinutes();`	Returns the hours or minutes of the time span, respectively.
`ts``.getTotalHours();`	Returns the total time in the time span as the real number of hours, such as 9.75 for (9 hours, 45 min).
`ts``.toString();`	Returns a string representation of the time span of hours and minutes, such as "28h46m".

The minutes should always be reported as being in the range of 0 to 59. That means that you may have to "carry" 60 minutes into a full hour. The following code creates a `TimeSpan` object and adds to it three times:

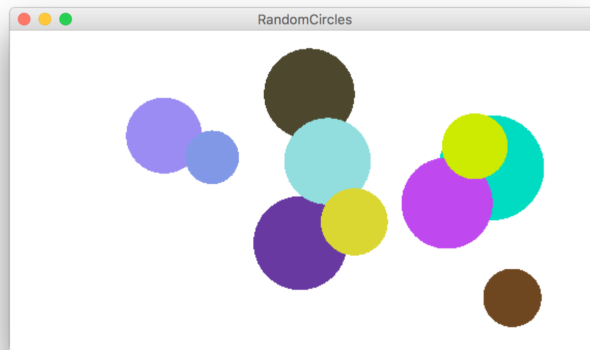
```
TimeSpan ts = new TimeSpan(1, 50);
ts.add(1, 55);
println(ts + " is " + ts.getTotalHours() + " hours");
// 3h45m is 3.75 hours
```

Notice that the printed time is not 2 hours, 105 min, although that's what you'd get by just adding field values.

10. 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.

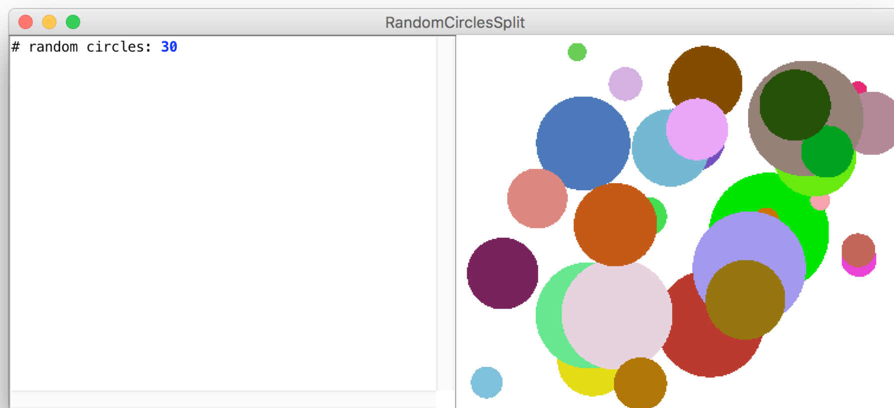
For this problem, write a **`GCanvas`** subclass **`RandomCirclesCanvas`** that implements similar behavior to the "Random Circles" problem from Section 3. As a quick refresher, that program drew **`N_CIRCLES`** random circles on the canvas, where each circle had a randomly chosen color, a randomly chosen 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();
      }
   }
}
```