Chris Piech
CS 106A

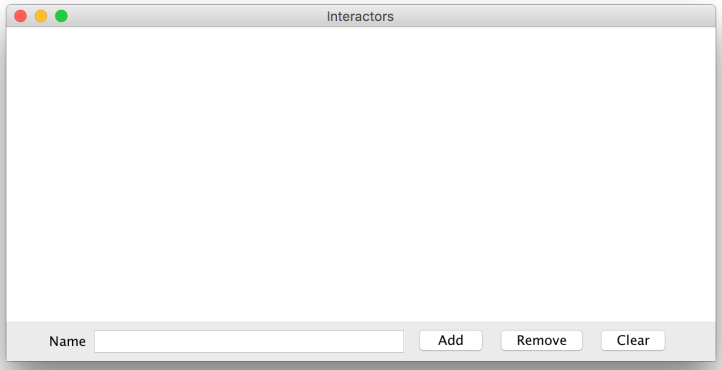
Section Handout 7
February 27, 2017

Section Handout #7: Interactors and Classes

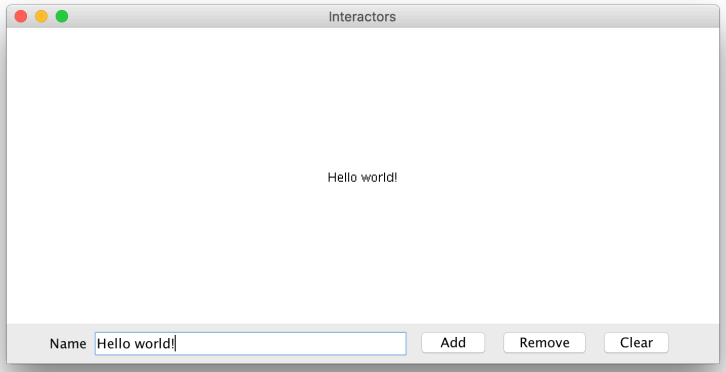
Portions of this handout by Eric Roberts and Nick Troccoli

1. Using Interactors

The purpose of this problem is to give you some practice using the kind of interactors you need for the NameSurfer project in Assignment #6. The specific example is to build an interactive design tool that allows the user to arrange labels in the window. Initially, the program presents an empty graphics canvas and a control bar containing a **`JLabel`**, a **`JTextField`**, and three **`JButton`**s:

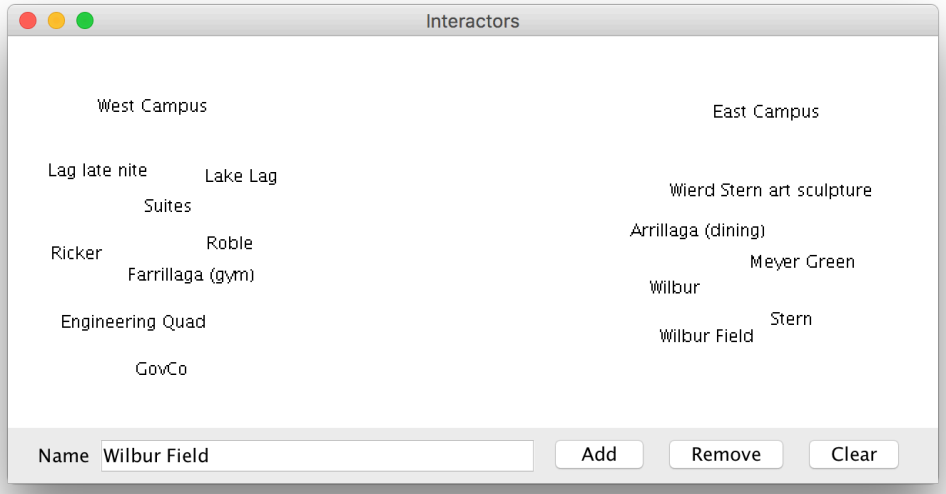


The most important operation in the program is to be able to add a new label to the screen, which you do by typing the name of the label in the **`JTextField`** and clicking the **`Add`** button. Doing so creates a new label with that text in the center of the window. For example, if you entered the string **`Hello world!`** in the **`JTextField`** and clicked **`Add`**, you would see the following result:



Once you have created a label, your program should allow the user to move the label around by dragging it with the mouse.

The ability to create new labels and drag them to new positions makes it possible to draw word diagrams or "word clouds" containing an arbitrary number of labels. For example, you could add more labels to make groups of some location names around campus:



The other two buttons in the control strip are **Remove** and **Clear**. The **Remove** button should delete the label whose name appears in the **`JTextField`**; the **Clear** button should remove all of the labels. Note that you do not need to worry about the user trying to create multiple labels with the same text–you can assume that all labels are unique.

While the operations in this program are conceptually simple, they influence the design in the following ways:

- The fact that you may need to remove a label by name forces you to keep track of the labels that appear in the window in some way that allows you to look up a label given the text that it displays. You therefore need some structure—and there is an obvious choice in the Java Collection Framework—that keeps track of all the labels on the screen.
- If the only objects in the window were labels, you could implement the **Clear** button by removing everything from the **`GCanvas`**. While that would work for this assignment, you might want to extend the program so if there were other objects on the screen that were part of the application itself, they would remain on the screen. In that case, you would want to implement **Clear** by iterating through the set of labels on the screen and removing each one. Remember, you can assume all label names are unique.

2. The `Employee` Class

Similar to the **`Student`** class example from Chapter 6 in the book, write a class definition for a class called **`Employee`**, which keeps track of the following information:

- The name of the employee
- The employee's job title
- The employee's annual salary

The first two fields should be set as part of the constructor, and it should not be possible for the client to change the employee name after that. For the job title and salary, your class definition should provide getters and setters that manipulate those fields. Your class should also implement a **`promote`** method that promotes an employee by adding "Senior" to the front of an employee's job title and doubling their salary. Below, we've included a sample program that uses the **`Employee`** class. Note that defining our own class makes it *much* easier to store information pertaining to each employee!

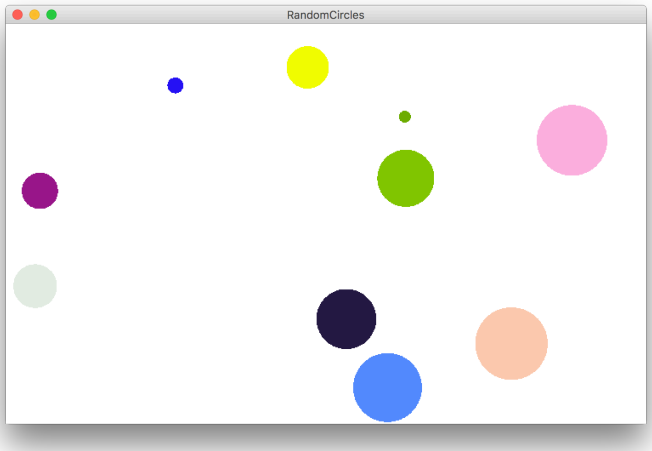
```
/**
 * File: EmployeeExample.java
 * This file contains a sample program using the Employee class.  It reads in
 * employee information until we read in the empty string, randomly promotes
 * one of the employees  entered, and then prints out all employees.
 */

import java.util.*;
import acm.program.*;
import acm.util.RandomGenerator;

public class EmployeeExample extends ConsoleProgram {

   public void run() {
      ArrayList<Employee> employees = readInEmployees();

      // Randomly promote a single employee
      int randomEmployeeNum = rgen.nextInt(employees.size());
      Employee employeeToPromote = employees.get(randomEmployeeNum);
      employeeToPromote.promote();
      println(employeeToPromote.getName() + " was promoted!\n\n");

      printEmployees(employees);
   }

   /*
    * Reads in a list of employees until the empty string is entered.
    * Returns an ArrayList of all Employees entered.
    */
   private ArrayList<Employee> readInEmployees() {
      ArrayList<Employee> employees = new ArrayList<Employee>();
      while (true) {
         String name = readLine("---\nName: ");
         if (name.equals("")) break;
         String title = readLine("Title: ");
         int salary = readInt("Salary ($): ");
```

```
            Employee newEmployee = new Employee(name, title);
            newEmployee.setSalary(salary);
            employees.add(newEmployee);
         }
         return employees;
      }

      /* Prints the name, title and salary for each of the given employees. */
      private void printEmployees(ArrayList<Employee> employees) {
         for (int i = 0; i < employees.size(); i++) {
            Employee currentEmployee = employees.get(i);
            println("--- " + currentEmployee.getName() + " (" +
                  currentEmployee.getTitle() + ") ---");
            println("Salary: $" + currentEmployee.getSalary());
         }
      }

      private RandomGenerator rgen = RandomGenerator.getInstance();
}
```

3. 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.

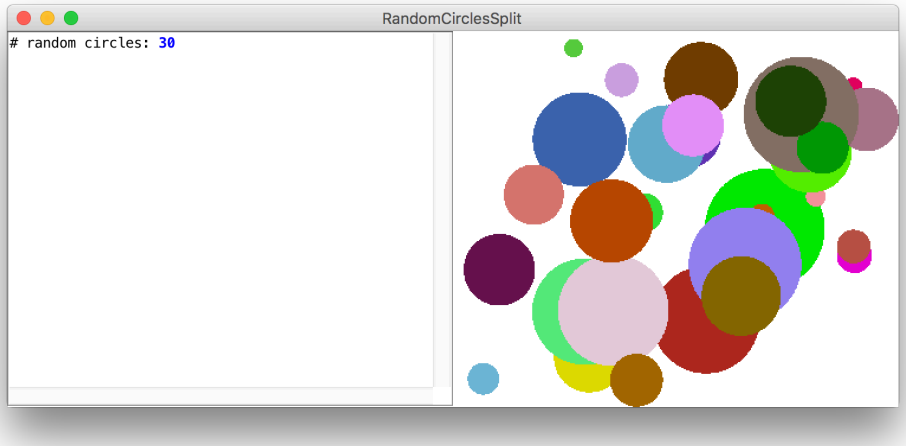
For this problem, write a **`GCanvas`** subclass **`RandomCirclesCanvas`** that implements similar behavior to the "Random Circles" problem from Section 3. As a quick refresher, that program drew **`N_CIRCLES`** random circles on the canvas, where each circle had a randomly chosen color, a randomly chosen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