Chris Piech
CS 106A

Section #7
May 23, 2018

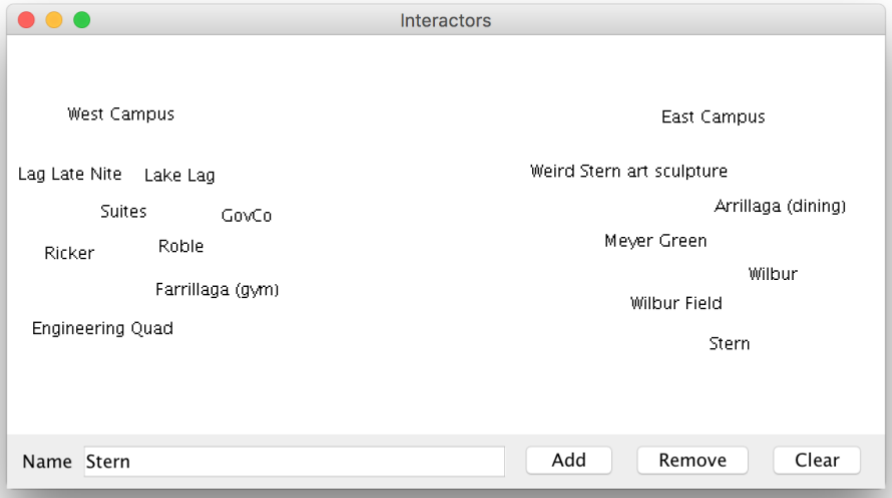
Section Handout #7: Interactors and Classes

Portions of this handout by Eric Roberts, Nick Troccoli, and Julia Daniel

Overview: these problems will give you practice with writing programs with interactors such as **`JLabel`**s, **`JTextField`**s and **`JButton`**s. 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 with little graphics-related code.

1. 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:



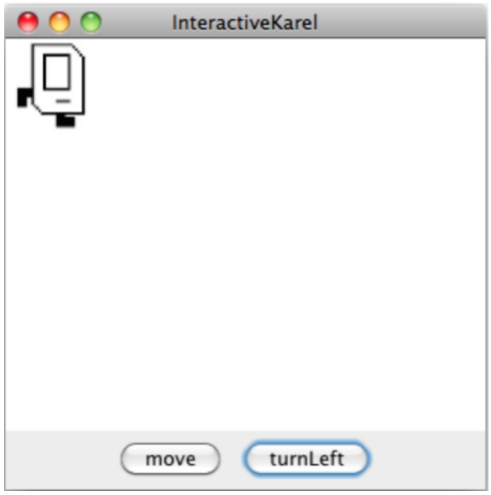
The program should have the following interactors: a "Name" label, a text field, and buttons to add, remove, and clear labels. To **add** a label to the screen, the user should be able to type text into the text field and click "Add" to add it. Labels should initially be added at the center of the screen. 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, how could we implement **clear**?

2. Interactive Karel

Write a complete program named `InteractiveKarel` with a graphical user interface that imitates some aspects of how Karel the Robot behaves.



As shown in the diagram above, you should create a **move** button and a **turnLeft** button. Pressing **move** causes Karel to move one length in the direction it is facing, and pressing **turnLeft** causes Karel to turn left. If moving Karel would cause it to move off the screen, just keep Karel where it is. You can assume you are given four images for each of the directions that Karel can face called `KarelEast.jpg`, `KarelWest.jpg`, `KarelSouth.jpg`, and `KarelNorth.jpg`. You can assume each of these images are 64 pixels in width and height. Karel should start facing east in the upper-left corner. Note: It may simplify your code to know that `GraphicsProgram`s come with pre-existing String constants named `NORTH`, `SOUTH`, `EAST`, and `WEST` whose values are `"North"`, `"South"`, etc.

3. 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 an employee's **name**, **job title** and **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.*;

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();
}
```

4. 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!