Eric Roberts
CS 106A

Handout #64A
March 10, 2010

Answers to Practice Final Examination #2

Problem 1—Short answer (15 points)

1a)

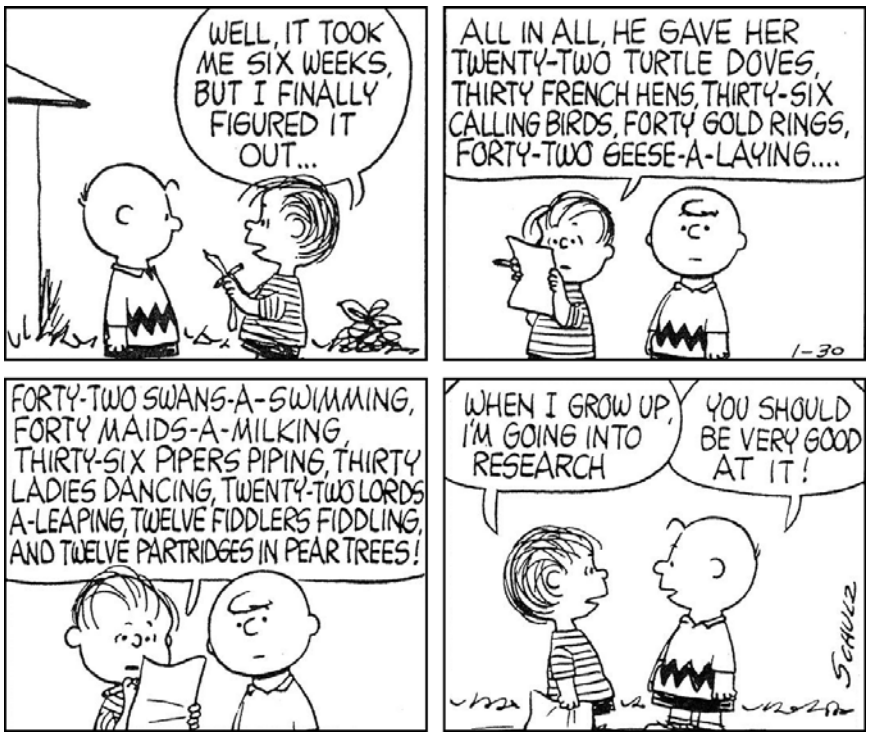
`array`

0	12	22	30	36	40	42	42	40	36	30	22	12
0	1	2	3	4	5	6	7	8	9	10	11	12

The values for this problem (which was taken from an autumn quarter final and thus came at the right season) indicate the total number of gifts in each category if you take the words to "The Twelve Days of Christmas" literally. At the end of true love's gift-giving spree, the total haul contains:

12 Partridges in pear trees
22 Turtle doves
30 French hens
36 Calling birds
40 Gold rings
42 Geese-a-laying
42 Swans-a-swimming
40 Maids-a-milking
36 Ladies dancing
30 Lords-a-leaping
22 Pipers piping
12 Drummers drumming

Charles M. Shultz offered a more interesting rendition of this problem in 1963:



1b)
1. Missing declaration and initialization of **`result`**
2. Illegal call to **`ch.isLetter()`** instead of **`Character.isLetter(ch)`**
3. Failed to set **`inWord`** to **`true`** when a letter was found

Problem 2—Using the `acm.graphics` library (15 points)

```
public class GraphicNim extends GraphicsProgram {

   public void init() {
      setupCoins();
      addMouseListeners();
   }

/** Called when the mouse is clicked */
   public void mouseClicked(MouseEvent e) {
      GObject obj = getElementAt(e.getX(), e.getY());
      if (obj != null) {
         int index = coinList.indexOf(obj);
         if (index >= Math.max(0, coinList.size() - 3)) {
            for (int i = coinList.size() - 1; i >= index; i--) {
               remove((GObject) coinList.get(i));
               coinList.remove(i);
            }
         }
      }
   }

/* Sets up the game */
   private void setupCoins() {
      double widthNeeded = COIN_COUNT * COIN_SIZE
                           + (COIN_COUNT - 1) * COIN_SEP;
      double x = (getWidth() - widthNeeded) / 2;
      double y = (getHeight() - COIN_SIZE) / 2;
      coinList = new ArrayList();
      for (int i = 0; i < COIN_COUNT; i++) {
         GOval coin = new GOval(x, y, COIN_SIZE, COIN_SIZE);
         coin.setFilled(true);
         coin.setFillColor(Color.GRAY);
         add(coin);
         coinList.add(coin);
         x += COIN_SIZE + COIN_SEP;
      }
   }

/* Private instance variables */
   private ArrayList coinList;

/* Private constants */
   private static final int COIN_COUNT = 11;
   private static final int COIN_SIZE = 20;
   private static final int COIN_SEP = 10;

}
```

Problem 3—Strings (15 points)

```
/** Returns true if s1 and s2 are anagrams of each other */
   private boolean isAnagram(String s1, String s2) {
      int[] table1 = createFrequencyTable(s1);
      int[] table2 = createFrequencyTable(s2);
      for (int i = 0; i < table1.length; i++) {
         if (table1[i] != table2[i]) return false;
      }
      return true;
   }

/** Creates a letter-frequency table for the string */
   private int[] createFrequencyTable(String str) {
      int[] letterCounts = new int[26];
      for (char ch = 'A'; ch <= 'Z'; ch++) {
         letterCounts[ch - 'A'] = 0;
      }
      for (int i = 0; i < str.length(); i++) {
         char ch = str.charAt(i);
         if (Character.isLetter(ch)) {
            letterCounts[Character.toUpperCase(ch) - 'A']++;
         }
      }
      return letterCounts;
   }
```

Problem 4—Arrays (10 points)

```
/**
 * Returns an image that is twice the size of the original in each
 * dimension.  Each pixel in the original is replicated so that
 * it appears as a square of four pixels in the new image.
 */
   private GImage doubleImage(GImage image) {
      int[][] oldPixels = image.getPixelArray();
      int height = oldPixels.length;
      int width = oldPixels[0].length;
      int[][] newPixels = new int[2 * height][2 * width];
      for (int i = 0; i < height; i++) {
         for (int j = 0; j < width; j++) {
            int pixel = oldPixels[i][j];
            newPixels[2 * i][2 * j] = pixel;
            newPixels[2 * i][2 * j + 1] = pixel;
            newPixels[2 * i + 1][2 * j] = pixel;
            newPixels[2 * i + 1][2 * j + 1] = pixel;
         }
      }
      return new GImage(newPixels);
   }
```

Problem 5—Building graphical user interfaces (15 points)

```
/*
 * File: SignMaker.java
 * --------------------
 * This program allows the user to compose an advertising
 * sign consisting of multiple lines of centered text.  The
 * user can set the font of each line independently.
 */

import acm.graphics.*;
import acm.program.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;

public class SignMaker extends GraphicsProgram {

   public void init() {
      lineInputField = new JTextField(CHARS_IN_LINE_FIELD);
      lineInputField.addActionListener(this);
      fontField = new JTextField(CHARS_IN_FONT_FIELD);
      fontField.setText("Times-Bold-24");
      currentBaseline = 0;
      add(new JLabel("Line: "), SOUTH);
      add(lineInputField, SOUTH);
      add(new JLabel("  Font: "), SOUTH);
      add(fontField, SOUTH);
   }

/**
 * Called when an action event occurs.
 */
   public void actionPerformed(ActionEvent e) {
      if (e.getSource() == lineInputField) {
         GLabel label = new GLabel(lineInputField.getText());
         label.setFont(fontField.getText());
         currentBaseline += label.getHeight();
         double x = (getWidth() - label.getWidth()) / 2;
         add(label, x, currentBaseline);
         lineInputField.setText("");
      }
   }

/* Instance variables */
   private int currentBaseline;
   private JTextField lineInputField;
   private JTextField fontField;

/* Constants */
   private static final int CHARS_IN_LINE_FIELD = 30;
   private static final int CHARS_IN_FONT_FIELD = 15;

}
```

Problem 6—Using Java collections (15 points)

```
import acm.util.*;
import java.io.*;
import java.util.*;

/**
 * This class localizes strings for an internationalized application.
 */
public class Localizer {

/**
 * Creates a new Localizer from the data in the specified file.
 */
   public Localizer(String filename) {
      map = new HashMap<String,String>();
      try {
         BufferedReader rd
           = new BufferedReader(new FileReader(filename));
         String word = null;
         while (true) {
            String line = rd.readLine();
            if (line == null) break;
            int equalSign = line.indexOf('=');
            if (equalSign == -1) {
               word = line;
            } else {
               String language = line.substring(0, equalSign);
               String translation = line.substring(equalSign + 1);
               map.put(word + "+" + language, translation);
            }
         }
         rd.close()
      } catch (Exception ex) {
         throw new ErrorException(ex);
      }
   }

/**
 * Looks up the localization of the English word.
 */
   public String localize(String word, String language) {
      String key = word + "+" + language;
      if (map.containsKey(key)) {
         return map.get(key);
      } else {
         return word;
      }
   }

/* Private instance variables */
   private HashMap<String,String> map;

}
```

Problem 7—Essay: Extensions to the assignments (10 points)

- You need to change the initialization code so that it installs superbricks in the brick layout with some probability. For example, you can use the expression

```
rgen.randomChance(SUPERBRICK_PROBABILITY)
```

  to decide whether to install a superbrick, and then define **SUPERBRICK_PROBABILITY** so that you get the desired prevalence of superbricks. If your code does decide to create a superbrick, all you need to do is call **setFillColor** instead of **setColor** to have the brick display itself correctly on the screen.

- The most important change is to redesign the data structure to keep track of multiple balls that are active simultaneously. Most implementations of Breakout stored the velocity of the ball in two instance variables named something like **vx** and **vy**. Tha approach is no longer sufficient, because SuperBreakout requires each ball to maintain its own velocity. The most straightforward way to make sure that each ball keeps track of that information is to make the ball a new object class. One strategy would be to define a **GBall** class that extends **GOval**, but also keeps track of the *x* and *y* components of its current velocity, exporting getters and setters for each. If, however, you are going to the trouble of making a separate **GBall** class, it probably makes even more sense to have **GBall** extend **GCompound** as described on page 332 of the text, because doing so makes it possible to have the *x* and *y* location of the ball correspond to its center rather than its upper left corner. That change tends to make simulating the physics much easier, but would require making more changes in the Breakout program code.

- You need to modify the code so that all of the methods that work with the ball—detecting collisions, bouncing off the walls, and so forth—take a **GBall** as an argument instead of working with values in instance variables as was possible in Assignment #3. It is no longer possible, for example, to have a single method

```
private GObject getCollidingObject()
```

  that returns the object with which the ball is colliding. There are now potentially several balls, and **getCollidingObject** needs to know which ball it is considering. Here, the most likely strategy is to rewrite **getCollidingObject** so that it takes a specific ball as its argument and then retrieves any object colliding with that particular ball.

- You need to add some structure to the program that can keep track of the balls in play. In all likelihood, this structure will be an **ArrayList<GBall>**.

- You need to change the play of each turn so that the program begins by adding one ball to the array list. At each time step in the simulation, your program needs to go through every ball in the list. For each ball, you need to check whether it has collided with anything and then update its position and velocity, just as you did for the single ball in the earlier version. If a ball falls off the bottom, you need to remove it from the array list. If the ball hits a superbrick (which you can tell by checking to see if its color and fill color are different), you need to add a new ball to the array list, exactly as you did at the beginning of the turn. A turn ends when the array list is empty or the last brick is removed.

- You need to determine what to do if two balls hit each other. To have them bounce as they would in physics is surprisingly tricky, because two balls can easily get "glued together" if you're not careful. It's easier on the whole to have the balls annihilate each other by removing both of them from the array list.