Eric Roberts
CS 106A

Handout #58
May 25, 2012

Practice Final Examination #1

Review session: **Wednesday, June 6, 7:30–9:30 P.M. (Hewlett 200)**

Scheduled finals: **Friday, June 8, 8:30–11:30 A.M. (Dinkelspiel Auditorium)**
Monday, June 11, 8:30–11:30 A.M. (Hewlett 200)

This handout is intended to give you practice solving problems that are comparable in format and difficulty to those which will appear on the final examination. A solution set to this practice examination will be handed out on Wednesday along with a second practice exam.

Time of the exam

The final exam is scheduled at two different times during exam period, as shown at the top of this handout. Please note that the June 8 final is in Dinkelspiel Auditorium rather than the regular classroom. You may take the exam at either of the two scheduled times and need not give advance notice of which exam you plan to take. If you are unable to take the exam at either of the scheduled times, please send an e-mail message to `eroberts@cs.stanford.edu` stating the following:

- The reason you cannot take the exam at the scheduled time.
- A three-hour period on Monday or Tuesday of exam week at which you could take the exam. This time must be during the regular working day and must therefore start between 8:30 and 2:00.

In order to take an alternate exam or to arrange special accommodations, I must receive a message from you by 5:00 P.M. on Friday, June 1. Replies will be sent by electronic mail by Monday, June 4.

Review session

The course staff will conduct a review session in Hewlett 200 from 7:30 to 9:30 P.M. on Wednesday, June 6. We will announce the winners of the Adventure Contest and hold the random grand-prize drawing at the beginning of the review session.

Coverage

The exam covers the material presented in class through next Wednesday's class, which means that you are responsible for the material in Chapters 1 through 13 of *The Art and Science of Java,* with the following exceptions:

- *Chapter 7.* There will not be any questions on the low-level representation of data as bits or any problems involving heap-stack diagrams.
- *Chapter 10.* In this chapter, you are responsible for the material in sections 10.1, 10.2, 10.3, 10.5, and 10.6. The only Swing interactors you will be expected to use are the ones from the NameSurfer assignment: `JLabel`, `JButton`, and `JTextField`.

- *Chapter 11.* You are responsible for all the material in this chapter except for the bit-manipulation operators in section 11.7. Image-manipulation problems that use no bit operations (such as the **`rotateLeft`** problem from Section #6) are fair game.
- *Chapter 12.* You are responsible for the general idea of searching and sorting, as presented in sections 12.1 and 12.2, including the binary search and selection sort algorithms. The coverage of data files will be limited to reading a file line by line.
- *Chapter 13.* You should understand how to use the **`ArrayList`**, **`HashMap`**, and **`TreeMap`** classes, but need not understand their implementation.

General instructions

The instructions that will be used for the actual final look like this:

> Answer each of the questions given below. Write all of your answers directly on the examination paper, including any work that you wish to be considered for partial credit.
>
> Each question is marked with the number of points assigned to that problem. The total number of points is 100. We intend that the number of points be roughly equivalent to the number of minutes someone who is completely on top of the material would spend on that problem. Even so, we realize that some of you will still feel time pressure. If you find yourself spending a lot more time on a question than its point value suggests, you might move on to another question to make sure that you don't run out of time before you've had a chance to work on all of them.
>
> In all questions, you may include methods or definitions that have been developed in the course, either by writing the **`import`** line for the appropriate package or by giving the name of the method and the handout or chapter number in which that definition appears.
>
> The examination is open-book, and you may make use of any texts, handouts, or course notes. You may not, however, use a computer of any kind.

Note: To conserve trees, I have cut back on answer space for the practice exams. The actual final will have much more room for your answers and for any scratch work.

Please remember that the final is <u>open-book.</u>
Friday, June 8, 8:30–11:30 P.M. (Dinkelspiel Auditorium)
Monday, June 11, 8:30–11:30 P.M. (Hewlett 200)

Problem 1—Short answer (10 points)

1a) Suppose that the integer array `list` has been declared and initialized as follows:

```
private int[] list = { 10, 20, 30, 40, 50 };
```

This declaration sets up an array of five elements with the initial values shown in the diagram below:

`list`

10	20	30	40	50

Given this array, what is the effect of calling the method

```
mystery(list);
```

if `mystery` is defined as:

```
private void mystery(int[] array) {
   int tmp = array[array.length - 1];
   for (int i = 1; i < array.length; i++) {
      array[i] = array[i - 1];
   }
   array[0] = tmp;
}
```

Work through the method carefully and indicate your answer by filling in the boxes below to show the final contents of `list`:

`list`

1b) When I wrote the program for counting trigraphs in lecture on May 18, I noted that sorting the trigraphs alphabetically was not really what you wanted. Given that the decryption strategy wants you to find the most common trigraphs, it would have been much more useful if the trigraphs could instead be displayed in order of how often they occurred. Jeremy and I set out to write a method that would do just that, using the strategy outlined in the comments:

```
/**
 * Prints out a map of strings to integers in ascending order
 * by the integer values.  The strategy is to go through each
 * of the key/count pairs and add that information to a new
 * "reverse map" that associates each count with a list of
 * the keys that appeared that many times.  (Because there
 * may be many keys that appear the same number of times, the
 * value in the reverse map needs to be a list of keys.)
 * Going through this map in order by count produces the
 * desired output.
 */
private void printAscendingByCount(Map<String,Integer> map) {
   Map<int,ArrayList<String>> reverseMap
        = new TreeMap<int,ArrayList<String>>();
   for (String key : map.keySet()) {
      int count = map.get(key);
      ArrayList<String> keyList = reverseMap.get(count);
      keyList.add(key);
      reverseMap.put(count, keyList);
   }
   for (Integer count : reverseMap.keySet()) {
      ArrayList<String> valueList = reverseMap.get(count);
      for (int i = 0; i < valueList.length; i++) {
         key = valueList.get(i);
         println(key + " = " + count);
      }
   }
}
```

Unfortunately, as clever as our strategy might be, the code is buggy, to the point that it doesn't even compile. Circle the bugs in the implementation and write a short sentence explaining the precise nature of each problem you identify.

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

Although the definition of filling for an arc (see page 314 in the text) is not necessarily what you would want for all applications, it turns out to be perfect for the problem of displaying a traditional pie chart. Your job in this problem is to write a method

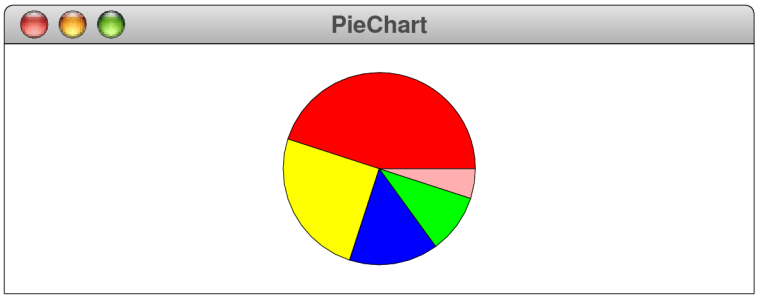
```
private GCompound createPieChart(double r, double[] data)
```

that creates a `GCompound` object for a pie chart with the specified data values, where `r` represents the radius of the circle, and `data` is the array of data values you want to plot.

The operation of the `createPieChart` method is easiest to illustrate by example. If you execute the `run` method

```
public void run() {
   double[] data = { 45, 25, 15, 10, 5 };
   GCompound pieChart = createPieChart(50, data);
   add(pieChart, getWidth() / 2, getHeight() / 2);
}
```

your program should generate the following pie chart in the center of the window:



The red wedge corresponds to the 45 in the data array and extends 45% around the circle, which is not quite halfway. The yellow wedge then picks up where the red wedge left off and extends for 25% of a complete circle. The blue wedge takes up 15%, the green wedge takes up 10%, and the pink wedge the remaining 5%.

As you write your solution to this problem, you should keep the following points in mind:

- The values in the array are not necessarily percentages. What you need to do in your implementation is to divide each data value by the sum of the elements to determine what fraction of the complete circle each value represents.
- The colors of each wedge are specified in the following constant array:

  ```
  private static Color[] WEDGE_COLORS = {
     Color.RED, Color.YELLOW, Color.BLUE, Color.GREEN,
     Color.PINK, Color.CYAN, Color.MAGENTA, Color.ORANGE
  };
  ```

  If you have more wedges than colors, you should just start the sequence over, so that the eighth wedge would be red, the ninth yellow, and so on.
- The reference point of the `GCompound` returned by `createPieChart` must be the center of the circle.

Problem 3—Strings (15 points)

A ***word-ladder puzzle*** is one in which you try to connect two given words using a sequence of English words such that each word differs from the previous word in the list only in one letter position. For example, the figure at the right shows a word ladder that turns the word **TEST** into the word **OVER** using eight single-letter steps.

```
T E S T
↓
B E S T
      ↓
B E E T
      ↓
B E E S
↓
B Y E S
↓
E Y E S
↓
E V E S
      ↓
E V E R
↓
O V E R
```

In this problem, your job is to write a program that checks the correctness of a word ladder entered by the user. (In CS 106B, you will learn how to write a program that finds word ladders.) Your program should read in a sequence of words and make sure that each word in the sequence follows the rules for word ladders, which means that each line entered by the user must

1. Be a legitimate English word
2. Have the same number of characters as the preceding word
3. Differ from its predecessor in exactly one character position

Implementing the first condition requires that you have some sort of dictionary available, which is well beyond the scope of a 15-minute problem. You may therefore assume the existence of a **Lexicon** class that exports a method called **contains** that checks whether a word is stored in that lexicon (which is just a fancy name for a list of words). You may also assume that a complete English dictionary has already been stored in an instance variable named **english**, which means that you can use the following test to determine whether **word** contains a valid English word:

```
if (english.contains(word)) . . .
```

If the user enters a word that is not legal in the word ladder, your program should print out an error message and let the user enter another word. It should stop when the user enters a blank line. Thus, your program should be able to duplicate the following sample run (the italicized messages don't appear but help to explain what's happening):

```
CheckWordLadder
Program to check a word ladder
Enter a sequence of words ending with a blank line
TEST
OVER                          You can't change all the letters at once
That word is not legal. Try again.
BEST
BEET
BEETS                         Each entry must be the same length
That word is not legal. Try again.
BEES
BXES                          All entries must be English words
That word is not legal. Try again.
BYES
EYES
EVES
EVER
OVER
                              Blank line denotes the end
```

Problem 4—Arrays (10 points)

If you spent any time at all playing your Yahtzee program, you know that it would be wonderful if there were a two-pair category for all the times you just missed getting a full house. Write a Java method

```
private boolean checkTwoPairCategory(int[] dice)
```

that takes an array of five die values and returns **`true`** if and only if the values in the **`dice`** array contain two pairs and a fifth value that does not match the others.

In writing your answer to this problem, you should keep the following points in mind:

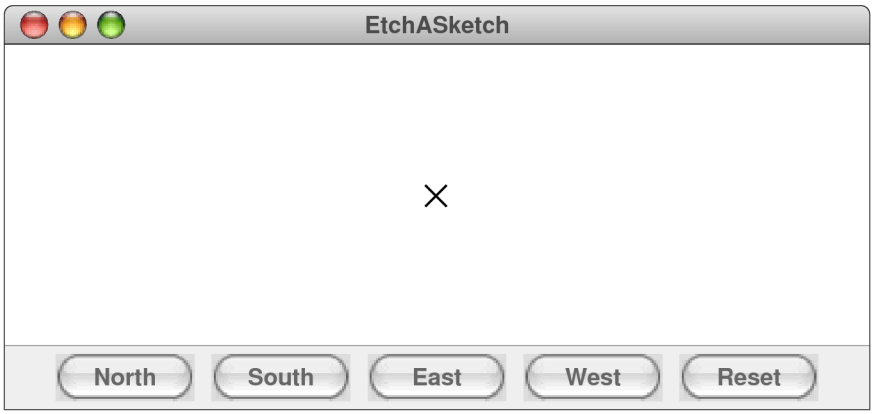
- You do not have to check that the array contains five elements or that the values in the array fall between 1 and 6.
- You may not change or reorder the values in the array you have been passed by the caller. You may, however, create temporary arrays as part of your implementation.
- A set of dice fits the two-pair category only when the paired values are different and the fifth value doesn't match any of the others.

Problem 5—Building graphical user interfaces (20 points)

> *I think you hit a reset button for the fall campaign. Everything changes. It's almost like an Etch A Sketch. You can kind of shake it up, and we start all over again.*
>
> —Eric Fehrnstrom, March 20, 2012

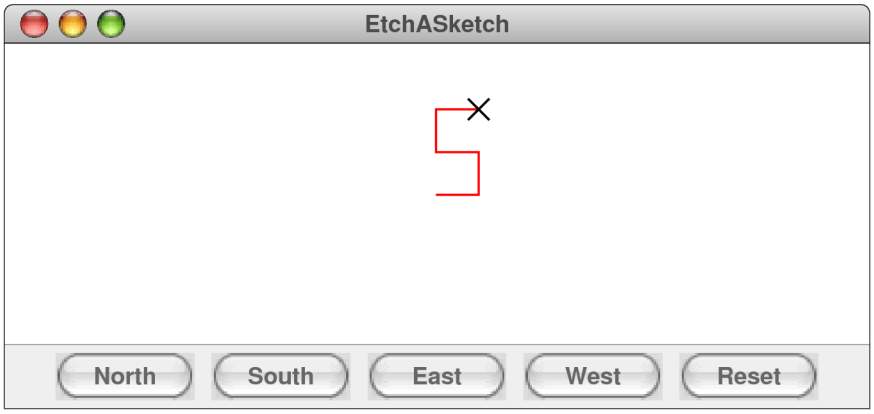
Write a **`GraphicsProgram`** that does the following:

1. Create a control strip with buttons labeled **North**, **South**, **East**, **West**, and **Reset**.
2. Create an `X`-shaped cross ten pixels wide and ten pixels high.
3. Adds the cross so that its center is at the center of the graphics canvas. Once you have completed these steps, the display should look like this:



4. Implement the actions for the button so that clicking on any of these buttons moves the cross 20 pixels in the specified direction. At the same time, your code should add a red **`GLine`** that connects the old and new locations of the pen.

Keep in mind that each button click adds a new **`GLine`** that starts where the previous one left off. The result is therefore a line that charts the path of the cross as it moves in response to the buttons. For example, if you clicked **East**, **North**, **West**, **North**, and **East** in that order, the screen would show a Stanford "S" like this:



Clicking the **Reset** button should remove all of the **`GLine`** segments and return the cross to the center of the window.

REUTERS

Morgan Stanley to adjust prices on Facebook trades

By Joseph A. Gainnone
NEW YORK | Wed May 23, 2012 1:22pm EDT

(Reuters) - Morgan Stanley told brokers on Wednesday it is reviewing every Facebook Inc trade and will make price adjustments for retail customers who paid too much during the social network company's debut last week, according to an internal memo.

Problem 6—Data structures (20 points)

In the wake of Facebook's troubled IPO, the Morgan Stanley brokerage that handled the offering has been forced to make restitution to some clients, primarily for late trades. Suppose, for example, that a client ordered a sale at 11:28am last Friday, when Facebook was selling at $40.00 a share. Given the many delays on that day, Morgan Stanley might not have been able to execute the sell order until 3:58pm, when Facebook shares had dropped to $38.07. That client therefore lost $1.93 per share, which adds up quickly if the trade involved a large block of shares.

Suppose that Morgan Stanley has hired you to write a simple application to calculate refunds due to its customers. They have given you a data file containing the complete history of the share price for Facebook in the early days of trading. Each line of the file contains a date/time string followed by an equal sign (typically surrounded by spaces, although those spaces are not required in the format) and then the share price as a floating-point value. That file therefore looks something like this:

FBSharePrice.txt

```
5/18/2012 11:30am = 42.0000
5/18/2012 11:31am = 42.0125
5/18/2012 11:32am = 42.0250
5/18/2012 11:33am = 42.0250
5/18/2012 11:34am = 40.9474
5/18/2012 11:35am = 40.8425
5/18/2012 11:36am = 40.1500
5/18/2012 11:37am = 40.0367
5/18/2012 11:38am = 40.0000
5/18/2012 11:39am = 40.0000
5/18/2012 11:40am = 40.0000
 . . .
5/18/2012 3:56pm = 38.1050
5/18/2012 3:57pm = 38.0997
5/18/2012 3:58pm = 38.0700
5/18/2012 3:59pm = 38.2599
5/18/2012 4:00pm = 38.2699
5/21/2012 9:30am = 36.4900
5/21/2012 9:31am = 36.0047
5/21/2012 9:32am = 35.5189
 . . .
```

Write a **ConsoleProgram** that reads this database into a suitable structure and then performs the following actions:

1. Reads in the date and time at which the customer ordered the sale.
2. Reads in the date and time at which the sale actually took place.
3. Reads in the number of shares involved.
4. Calculates the refund due to the customer if the sale had occurred at the right time.
5. Prints out the refund value or a message **"No refund due"**.

For example, your program should be able to duplicate the following sample run:

The program computes the difference in share costs (as discussed for this example on the preceding page) and then multiplies the $1.90 per share refund by 1000 to get the total refund due.

For Section #7, you wrote a **CalendarDate** class to represent a date. The Java libraries already include a **Date** class that will come in very handy here, particularly if we provide a conversion method for you to use as a tool. The following method converts a string into the **Date** object it represents:

```
private Date stringToDate(String str) {
   try {
      return new SimpleDateFormat("MM/dd/yyyy hh:mma").parse(str);
   } catch (ParseException ex) {
      throw new ErrorException(ex);
   }
}
```

For example, if you call **stringToDate("5/18/2012 11:40am")**, you get a **Date** object representing that date and time. You can compare that object to other dates using **equals** and **compareTo** or use it as a key in a map.

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

In Crowther's original version of Adventure, the solution to one of the puzzles involves a magic word that automatically returns a treasure from wherever it happens to be to its initial location. In this problem, your job is to discuss the changes that would be necessary in the Adventure implementation to implement a similar feature in the game from Assignment #7. This question requires an essay answer, and you should not feel compelled to write any actual code unless you feel that doing so is the best way to convey your ideas. You should, however, indicate by name what classes and methods need to change.

To make this question more concrete, suppose that you have been asked to add a command

> **`ZAP`** *object*

that has the effect of returning the specified object to its initial room. Like **`LOOK`**, **`QUIT`**, **`INVENTORY`**, and the other action commands, the **`ZAP`** command can be executed in any room. Unlike the **`TAKE`** command, however, the object is not ordinarily in that room, but can be anywhere in the cave, including the player's inventory.

Before you get the idea that this question is too easy, note that there is no way in the current design to determine where an object is currently located. Given a room, you can find the objects in that room, and you have presumably added an array list somewhere to keep track of the player's inventory. Adding the object to the list in its starting room is relatively easy, but taking it out of its current location is hard. You need to think about the changes you need to make in the overall design to implement this feature.

In writing your solution, you should keep the following points in mind:

- You are not required to perform any error-checking on the format of the file. You do, however, need to include the **`try/catch`** statements that are required when you open or read from a file.
- You do not have to worry about formatting the refund value. Any number of digits after the decimal point is fine.
- Morgan Stanley is not likely to ask customers to return money if they did better as a result of the delay. Thus, if your program computes a refund that is zero or negative, it should simply report **`"No refund due"`**.