Mehran Sahami
CS106A

Handout #47
December 5, 2012

Practice Final Examination

Final Exam Time: Friday, December 14th, 12:15pm to 3:15pm

Final Exam Location by last name:

Last name starts with A - H: Hewlett 200
Last name starts with I - Z: Memorial Auditorium (ground floor)

Portions of this handout by Eric Roberts

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 exam.

Final Exam is open book, open notes, closed computer

The examination is open-book (specifically the course textbook *The Art and Science of Java* and the Karel course reader) and you may make use of any handouts, course notes/slides, printouts of your programs or other notes you've taken in the class. You may not, however, use a computer of any kind (i.e., you cannot use laptops on the exam).

Coverage

The final exam covers the material presented throughout the class (with the exception of the Karel material), which means that you are responsible for Chapters 1 through 13 of the class textbook *The Art and Science of Java*. You will not be responsible for the "standard Java" material (i.e., the `main` method, JAR files) nor the "advanced topics" on Threads covered in class on November 28th and 30th.

General instructions

Answer each of the questions included in the exam. 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 180. We intend for the number of points to be roughly comparable to the number of minutes you should spend on that problem.

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 textbook chapter number in which that definition appears.

Unless otherwise indicated as part of the instructions for a specific problem, comments will not be required on the exam. Uncommented code that gets the job done will be sufficient for full credit on the problem. On the other hand, comments may help you to get partial credit if they help us determine what you were trying to do.

Blank pages for solutions omitted in practice exam (but will be available on real exam)

In an effort to save trees, the blank pages that would be provided in a regular exam for writing your solutions have been omitted from this practice exam.

Problem 1: Short answer (15 points)

1a. We learned that when you pass an object as a parameter into a method, changes that are made to the object persist after the method completes execution. However, if you pass in an `int` as a parameter and change the value of that parameter in a method, the original `int` variable that was passed in remains unchanged. Explain why that is.

Answer for 1a:

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

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

This statement sets up an array of five elements with the initial values shown 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:

```
public 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`:

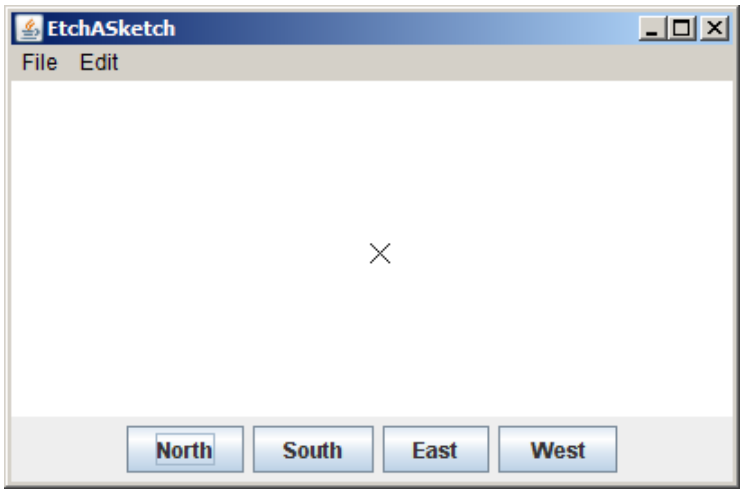
Answer to 1b:

`list`

Problem 2: Graphics and Interactivity (35 points)

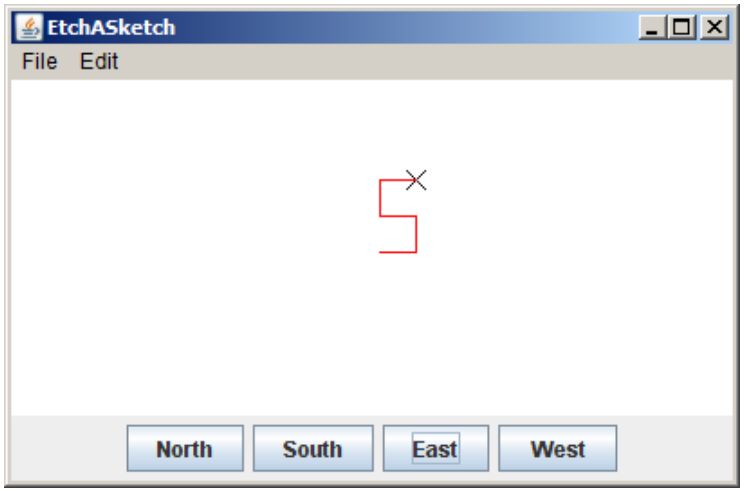
Write a **`GraphicsProgram`** that does the following:

1. Add buttons to the **South** region labeled **`"North"`**, **`"South"`**, **`"East"`**, and **`"West"`**.
2. Create an X-shaped cross 10 pixels wide and 10 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:

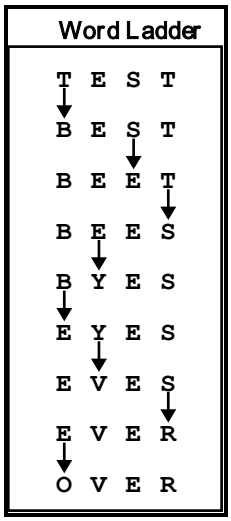


4. Implement the actions for the button so that clicking on any of these buttons moves the cross 20 pixels in the specified direction. At the same time, your code should add a **red** **`GLine`** that connects the old and new locations of the pen.

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 (note the "S" would be red, even though it does not appear so in the black and white handout):



Problem 3: Files and Strings (35 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.

In this problem, your job is to write a program that checks the correctness of a word ladder entered by the user. 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 beyond the scope of this problem. You may therefore assume the existence of a `Lexicon` class (generally speaking, a *lexicon* is simply a list of words) that exports the following method:

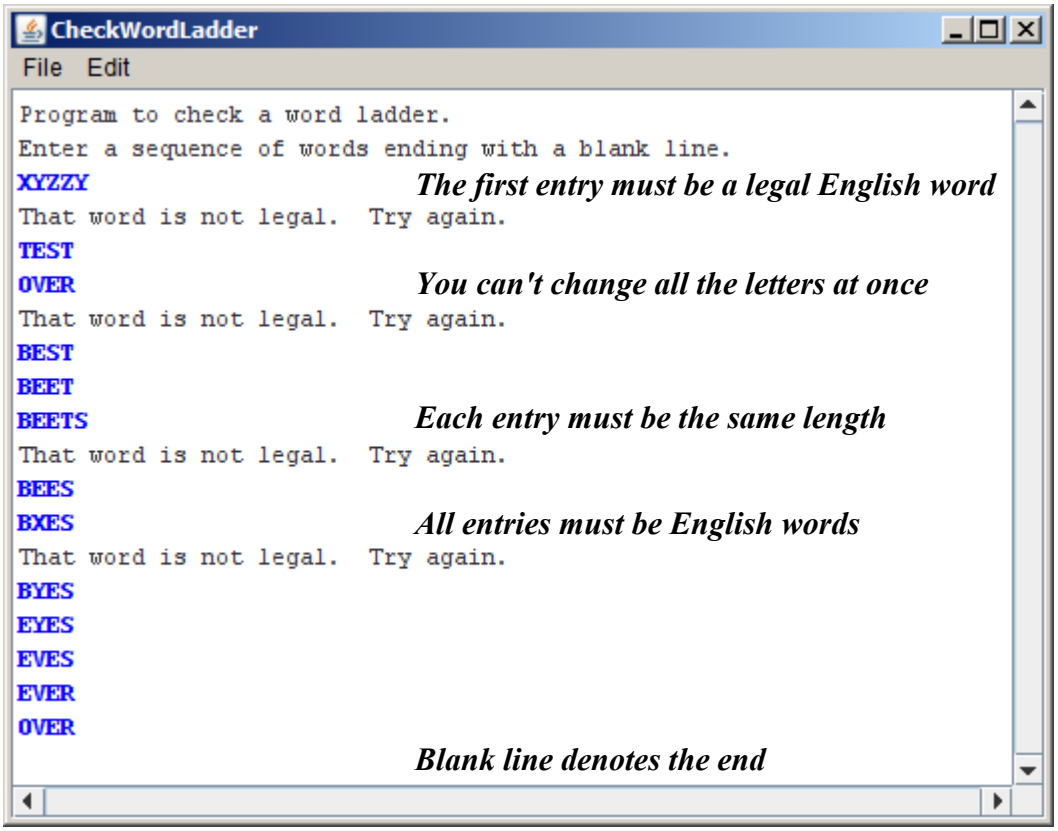
```
public boolean isEnglishWord(String str)
```

which takes a word (`String`) and returns `true` if that word is in the lexicon (i.e., the string passed is a valid English word). You may also assume that you have access to such a dictionary via the following instance variable declaration:

```
private Lexicon lexicon = new Lexicon("english.dat");
```

All words in the lexicon are in **upper case**.

If the user enters a word that is not legal in the word ladder, your program should print out a message to that effect and let the user enter another word. It should stop reading words when the user enters a blank line. Thus, your program should be able to duplicate the following sample run that appears on the next page (the italicized messages don't appear but are there to explain what's happening).


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

Problem 4: Arrays (20 points)

In the last several years, a new logical puzzle from Japan called *Sudoku* has become quite a hit in Europe and the United States. In Sudoku, you start with a 9x9 grid of numbers in which some of the cells have been filled in with digits between 1 and 9, as shown in the upper right diagram.

		7	4		3		2	9
	3	8		2		7		4
4	9		5		7	1	3	
8	4		9	7	6	3	1	2
7	2		3			9		6
	6	9		1	4		8	
	1	6		4	2	8		3
2			6		1			5
9		4	8		5	2	6	1

Your job in the puzzle is to fill in each of the empty spaces with a digit between 1 and 9 so that each digit appears exactly once in each row, each column, and each of the smaller 3x3 squares. Each Sudoku puzzle is carefully constructed so that there is only one solution.

Suppose that you wanted to write a program to check whether a proposed solution was in fact correct. Because that task is too hard for an array problem on an exam, your job here is simply to check whether the upper-left 3x3 square contains each of the digits 1 through 9. In the completed example (shown in the bottom right diagram), the 3x3 square in the upper left contains exactly one instance of each digit and is therefore legal.

1	5	7	4	8	3	6	2	9
6	3	8	1	2	9	7	5	4
4	9	2	5	6	7	1	3	8
8	4	5	9	7	6	3	1	2
7	2	1	3	5	8	9	4	6
3	6	9	2	1	4	5	8	7
5	1	6	7	4	2	8	9	3
2	8	3	6	9	1	4	7	5
9	7	4	8	3	5	2	6	1

If, however, you had made a mistake filling in the puzzle and come up with the following

1	5	7
6	3	8
4	9	6

instead, the solution would be invalid because this square contains two instances of the value 6 (and no instances of the value 2).

Your task in this problem is to write a method

```
private boolean checkUpperLeftCorner(int[][] matrix)
```

which looks at the 3x3 square in the upper left corner of the matrix and returns **`true`** if it contains one instance of each of the digits from 1 to 9. If it contains an integer outside of that range or contains duplicated values, **`checkUpperLeftCorner`** should return **`false`**.

In writing your solution, you may assume that the variable passed in as the **`matrix`** parameter has already been initialized as a 9x9 array of **`int`**s. You are also completely free to ignore the values outside of the 3x3 square in the upper left. Those values will presumably be checked by other code in the program that you are not responsible for.

Problem 5: Data structure design (25 points)

The **java.util** package contains a fairly large number of classes and interfaces that support collections of objects in one form or another. Collectively, these classes and interfaces are called the **Java collections framework.** In CS 106A, you've had the chance to work with the **ArrayList** class, which implements a more general **List** interface, and the **HashMap** class, which implements the **Map** interface.

Another useful interface in the Java collections framework is the **Queue** interface, which models a collection in which objects are added at one end and removed from the other, much as in a waiting line. We can consider a queue of **String**s, where the fundamental operations are **add**, which adds an **String** to the end of the queue, **poll**, which removes and returns the **String** at the front of the queue (or **null** if the queue is empty), and **size**, which returns the number of **String**s in the queue. These three methods define the **MinimalStringQueue** shown in Figure 1.

Your task in this problem is to write an implementation of the class **StringQueue**, which implements the **MinimalStringQueue** interface by storing **String**s in an **ArrayList** maintained as an instance variable inside the class.

As an illustration of how **StringQueue** works, you could create an empty queue by executing the statement:

```
StringQueue queue = new StringQueue();
```

You could then add the three ghosts from Dickens's *A Christmas Carol* like this:

```
queue.add("Christmas Past");
queue.add("Christmas Present");
queue.add("Christmas Future");
```

At this point, calling **queue.size()** should return 3 because there are three entries in the queue. The first call to **queue.poll()** would return **"Christmas Past"**, the second would return **"Christmas Present"**, and the third would return **"Christmas Future"**. If you called **queue.poll()** a fourth time, the return value would be **null**.

Figure 1: MinimalStringQueue interface

```
/**
 * This interface represents a collection of objects called a
 * "queue" in which new Strings are added at the end of the
 * queue and removed from the front, giving rise to a typical
 * first-come/first-served waiting line.
 */

public interface MinimalStringQueue {

/** Adds a new String to the end of the queue */
   public void add(String str);

/** Removes and returns the first String (or null if queue is empty) */
   public String poll();

/** Returns the number of entries in the queue. */
   public int size();

}
```

Problem 6: Java programming (30 points)

> *Q: What do you call Enron corporate officers who contributed money to Senators on both the left **and** the right?*
> *A: Ambidextrous scallywags.*
>
> —Steve Bliss, posting to the Googlewhacking home page

The Google™ search engine (which was developed here at Stanford by Larry Page and Sergey Brin) has rapidly become the search engine of choice for most users of the World Wide Web. A few years ago, it also gave rise to a pastime called *Googlewhacking* that quickly became quite popular among web surfers with far too much time on their hands. The goal of the game is to find a pair of English words so that both appear on exactly one Web page in Google's vast storehouse containing billions of pages. For example, before they were listed on the Googlewhacking home page, there was only one web page that contained both the word *ambidextrous* and the word *scallywags*.

Suppose that you have been given a method

```
public String[] googleSearch(String word)
```

that takes a single word and returns an array of strings containing the URLs of all the pages on which that word appears. For example, if you call

```
googleSearch("scallywags")
```

you would get back a string array that looks something like this:

`http://www.scallywags.ca/`
`http://www.effect.net.au/scallywags/`
`http://www.scallywags1.freeserve.co.uk/`
`http://www.scallywagsbaby.com/`
`http://www.sfsf.com.au/ScallywagsCoaches/`
`http://www.theatlantic.com/unbound/wordgame/wg906.htm`
`http://www.maisemoregardens.co.uk/emsworth.htm`

Each of the strings in this array is the URL for a page that contains the string *scallywags*. If you were to call

```
googleSearch("ambidextrous")
```

you would get a different array with the URLs for all the pages containing *ambidextrous*.

Your job in this problem is to write a method

```
public boolean isGooglewhack(String w1, String w2)
```

that returns `true` if there is exactly one web page containing both `w1` and `w2`. It should return `false` in all other cases, which could either mean that the two words never occur together or that they occur together on more than one page. Remember that you have the `googleSearch` method available and therefore do not need to write the code that actually scans the World Wide Web (thankfully!).

Problem 7: Using data structures (20 points)

This quarter you have also gotten experience with the HashMap class in Java. When working with HashMaps, sometimes cases arise where we wish to determine if two HashMaps have any key/value pairs in common. For example, we might have the following two Hashmaps (named **`hashmap1`** and **`hashmap2`**, respectively) that map from **`String`** to **`String`** (i.e., their type is **`HashMap<String,String>`**) and we want to count how many key/value pairs they have in common.

`hashmap1`

Key	Value
Alice	Healthy
Mary	Ecstatic
Bob	Happy
Chuck	Fine
Felix	Sick

`hashmap2`

Key	Value
Mary	Ecstatic
Felix	Healthy
Ricardo	Superb
Tam	Fine
Bob	Happy

In the example above, these two HashMaps have **two** key/value pairs in common, namely: "Mary"/"Ecstatic" and "Bob"/"Happy". Note that although the key "Felix" is in both HashMaps, the associated value with this key is different in the two maps (hence this does not count as a key/value *pair* that is common to both HashMaps). Similarly, just having the same value without the same key (such as the value "Fine" which is mapped to by different keys in the two different HashMaps) would also not count as a common key/value pair between the two HashMaps.

Your job is to write a method:

```
public int commonKeyValuePairs(HashMap<String,String> map1,
                               HashMap<String,String> map2)
```

that is passed two objects of type **`HashMap<String,String>`** and returns the number of common key/value pairs between the two HashMaps.