Practice Second Midterm Exam #1

Based on a handout by Eric Roberts and Mehran Sahami

This handout is intended to give you practice solving problems that are comparable in format and difficulty to those which will appear on the second midterm final exam.

Second Midterm Exam is Open Book, Open Notes, Closed Computer

The examination is open-book (specifically the course textbook *The Art and Science of Java*) 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 second midterm 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 are also responsible for the material presented in the lecture on graphs and collections.

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 120. 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 (10 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`:

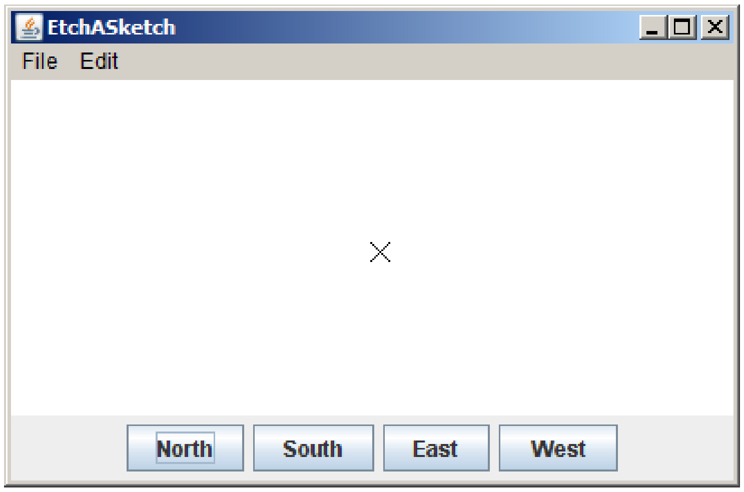
Answer to 1b:

`list`

Problem 2: Graphics and Interactivity (20 points)

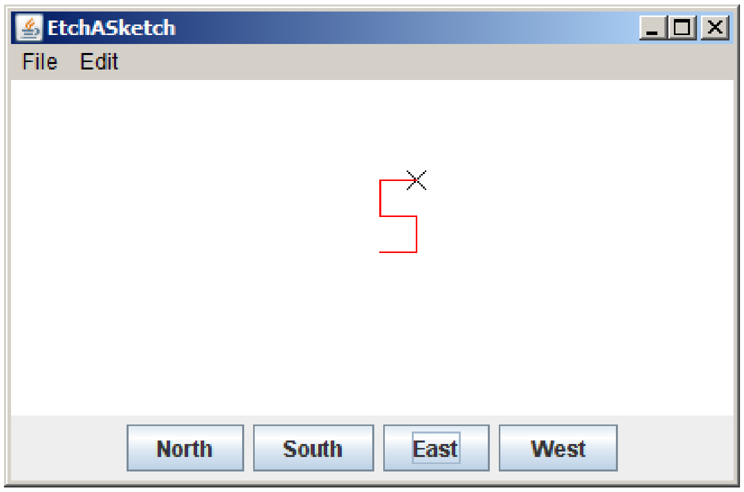
Write a **`GraphicsProgram`** that does the following:

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:

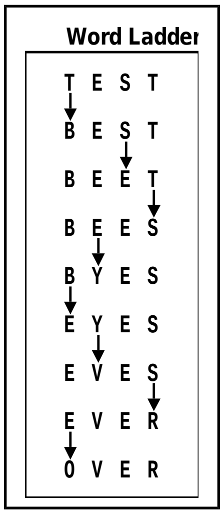


4. Implement the actions for the button so that clicking on any of these buttons moves the cross 20 pixels in the specified direction. At the same time, your code should add a **red** **`GLine`** that connects the old and new locations of the pen.

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 (20 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:

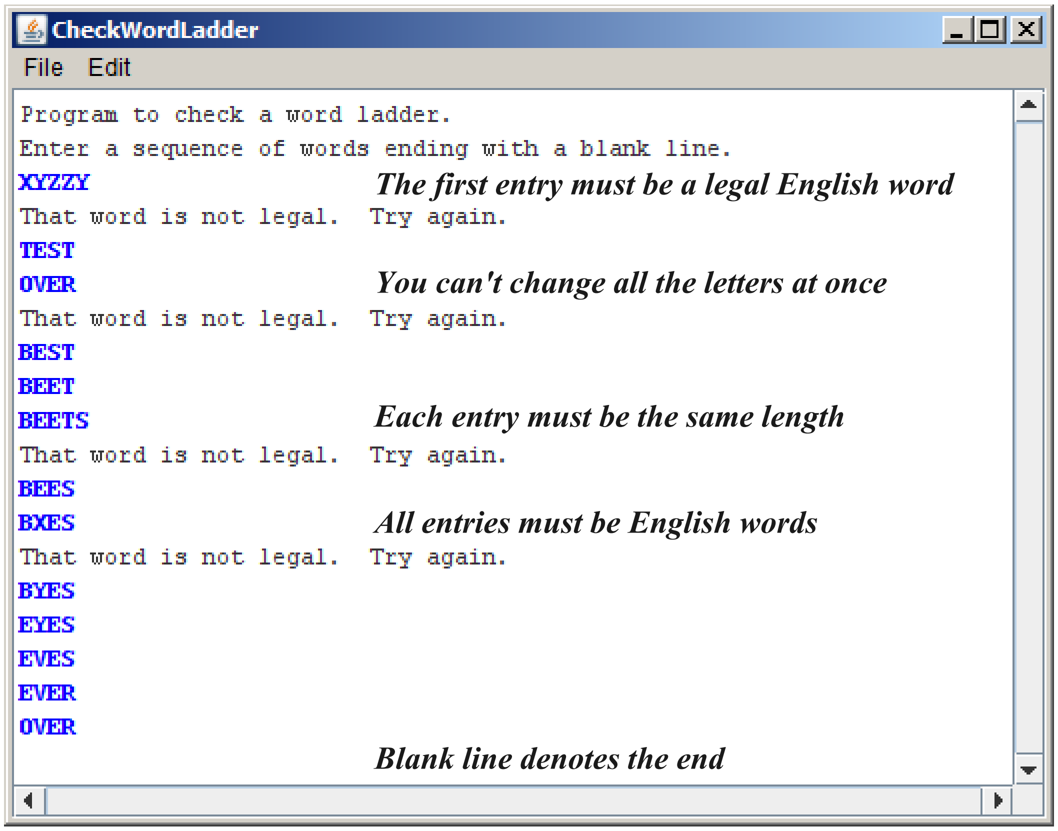
```
public boolean isEnglishWord(String str)
```

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).



```
Program to check a word ladder.
Enter a sequence of words ending with a blank line.
XYZZY                     The first entry must be a legal English word
That word is not legal.  Try again.
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 (25 points)

A *magic square* is an $n \times n$ grid of numbers with the following properties:

1. Each of the numbers 1, 2, 3, …, n^2 appears exactly once, and
2. The sum of each row and column is the same.

For example, here is a 3×3 magic square, which uses the numbers between 1 and $3^2 = 9$:

4	9	2
3	5	7
8	1	6

and here is a 5×5 magic square, which uses the numbers between 1 and $5^2 = 25$:

11	18	25	2	9
10	12	19	21	3
4	6	13	20	22
23	5	7	14	16
17	24	1	8	15

Write a method

```
private boolean isMagicSquare(int[][] square, int n);
```

that accepts as input a two-dimensional array of integers (which you can assume is of size $n \times n$) and returns whether or not it is a magic square.

Problem 5: Java programming (25 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 6: Using data structures (20 points)

This quarter you have also gotten experience with the **`HashMap`** class in Java. When working with **`HashMap`** s, sometimes cases arise where we wish to determine if two **`HashMap`** s have any key/value pairs in common. For example, we might have the following two **`HashMap`** (named **`hashmap1`** and **`hashmap2`**) that map from **`String`**s to **`String`**s (that is, they are **`HashMap<String,String>`**s) 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 **`HashMap`** s have **<u>two</u>** key/value pairs in common, namely: "Mary"/"Ecstatic" and "Bob"/"Happy". Note that although the key "Felix" is in both **`HashMap`**s, 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 **`HashMap`** s). 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 **`HashMap`**s) would also not count as a common key/value pair between the two **`HashMap`** s.

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.