Problem Set 3

I. Overview

Electronic submission of this assignment must be done through our course website. Be sure to use the precise file names indicated in each problem.

Begin by looking over chapters 6 and 10 in the Reges & Stepp textbook; it deals with exception-handling, file I/O and *ArrayLists*.

Unless otherwise indicated, everything that the human would type to the computer has been underlined. Everything the computer outputs is not.

II. Pencil-and-Paper Exercises (15—20 points)

Submit the answer to each problem in the named file. The file must be a plain "text" file (not a Word file, an RTF file, or any other non-text-file format.

[1] 4 points *use file Prob1.java*

Write a simple Java program named **`Prob1`** that outputs all of its *command-line arguments*, one per line, in *reverse order.* Here is an example to illustrate:

```
% java Prob1 fee fie foe fum foobar!
The 5 command-line args are:

foobar!
fum
foe
fie
fee
```

[2] 3 points *use file Prob2.txt*

Provide a simple specific example illustrating how a method might throw a **`nullPointerException`**.

Give a simple explanation of why the **`nullPointerException`** is NOT a "checked" exception.

[3] 3 points *use file Prob3.txt*

Suppose we have constructed an **`ArrayList`** of *`String`* values, named **`a`**. For example, **a** might contain just 3 values, and the output by executing the statement **`System.out.println (a) ;`** could look like this

```
[Now, Later, Ever]
```

Consider now the following incorrect loop which is supposed to add the string "NOT" in front of each word in the **`ArrayList`**:

```
for (int i = 0; i < a.size(); i++)  a.add (i, "NOT");
```

If this loop worked correctly, then the output by executing a **`System.out.println (a);`** would be

```
[NOT, Now, NOT, Later, NOT, Ever]
```

Unfortunately, this loop never terminates! Why not? What is a very simple way of fixing the loop so that it works correctly?

[4] 5 points *use file Prob4.java*

Consider the following snippet of code that asks the user to input 2 integers and then divides them:

```
Scanner keyboard = new Scanner (System.in);
int n1, n2;

System.out.print("Type an int: ");
n1 = keyboard.nextInt();
System.out.print("Now type another int: ");
n2 = keyboard.nextInt();
int r =  n1 / n2;
```

Now place the code beginning with the first **System.out** into a **try-catch** block with multiple catches so that different error messages are printed if

- we attempt to divide by 0; or
- the user enters textual data instead of integers (`java.util.InputMismatchException`)

If either of these conditions should occur, the program should print a polite error message, and then loop back to allow the user to enter new data. Do NOT completely rewrite the code; just add the requested elements. Your code should NOT include a **throw** statement.

[5] 5 points ***(for extra credit)*** *use file ExcExample.java*

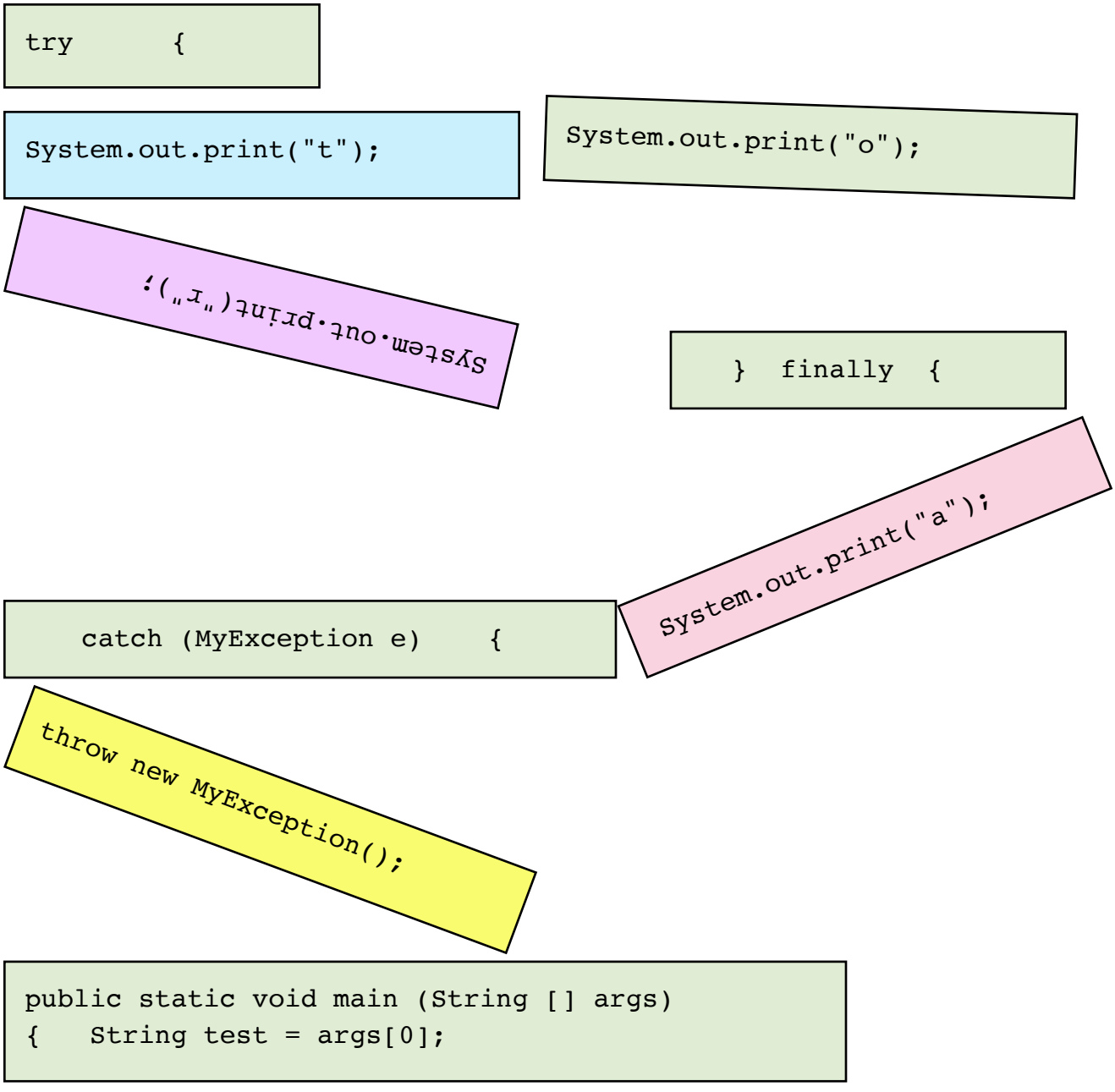
A poorly-written, but working Java program got scrambled up on the fridge with each statement on a separate kitchen magnet. When the code was in the correct order, here's what happened at execution time:

```
hleitner% java ExcExample Warren
thaws

hleitner% java ExcExample Foobar
throws
```

Note that the program was run two different times with a different command-line argument in each case.

Reconstruct the Java code by placing the code magnets in the correct order. Note that some of the curly braces fell on the floor, and are too small to pick up — so feel free to add as many of those as you need!



```
doRisky (test);
```

```
if (“Warren".equals(arg))
```

```
System.out.print("w");
```

```
System.out.println("s");
```

```
static void doRisky (String arg) throws MyException
  {   System.out.print ("h");
```

```
class MyException extends Exception {}
public class ExcExample {
```

III. Programming Problems (40—65 points total)

[6] 40 points *use file ExamAnalysis.java*

A large number of student responses to questions on a multiple-choice examination are to be processed by a program that you will write. The input to your program will consist of

⇒ the correct answers to the exam questions;

⇒ the actual student responses (assume that there can be no more than 100 students). The responses are contained in a file; we will supply a sample file for you to use.

Note that for every exam question there are precisely 6 possible answers: **A**, **B**, **C**, **D**, **E** and *blank.* The last case is to allow the possibility of a student not responding at all to a question.

Since the student answers are to be read in from a *file*, you should "echo" the data on to the computer display screen as it is being read. You should assume that the user of your program knows nothing about how computers work, so your program needs to be friendly and easy to operate.

The output from your program will consist of

⇒ a complete listing of the students' scores (the number of correct answers, incorrect answers, and blank answers); these may be given by student number (as illustrated below).

⇒ a second listing, like the one shown below, which illustrates how the group as a whole answered each question on the examination.

Here is how the output from your program should appear when executed:

```
I hope you are ready to begin ...

Please type the correct answers to the exam questions,
  one right after the other:   ABCEDBACED
```

```
What is the name of the file containing each student's
 responses to the 10 questions?  exams.dat

Student #1's responses:  ABDEBBAC D
Student #2's responses:  ABCE CACED
Student #3's responses:    DCE AEDC
Student #4's responses:  ABCEB ACED
Student #5's responses:  BBCEDBACED
Student #6's responses:  DBCE CACED
Student #7's responses:  ABCE CA E          (note the space after the final E)
Student #8's responses:  BBE  CACED
Student #9's responses:  CBCEDBACED
We have reached "end of file!"

Thank you for the data on 9 students.  Here's the analysis:
```

Student #	Correct	Incorrect	Blank
1	7	2	1
2	8	1	1
…	…	…	…
…	…	…	*{ the whole table should be filled in}*
…	…	…	…
9	9	1	0

```
QUESTION ANALYSIS   (* marks the correct response)
~~~~~~~~~~~~~~~~~

Question #1:

   A*       B       C       D       E       Blank

   4        2       1       1       0        1

 44.4%    22.2%   11.1%   11.1%    0.0%    11.1%


Question #2:

   A        B*      C       D       E       Blank
    …
    …
```

etc. Again, be sure to fill in the entire table for all 10 questions

Be sure to test your program out on the sample data provided above which corresponds to an exam having 10 multiple-choice questions with just 9 students taking the exam. **But remember — your program should be general enough that it would work with an exam having any number (up to 100) different multiple-choice questions and with up to 100 different students taking the exam.**

You may wish to copy the sample student answers to the nine questions from the file **`exams.dat`** into your own directory. You can find this file in the Unit 5 Lecture Files section of the Java Resources page

[7] 25 points (for graduate-credit students only) *use file **CaesarCipher.java***

Undergraduate-credit students may attempt this problem for "extra credit."

A *Caesar Cipher* is one of the earliest forms of an alphabetic cipher for creating "secret messages." A Caesar Cipher works by substituting for each letter in the original "plaintext" a letter obtained by shifting the alphabet by a constant number.

For example, a Caesar Cipher with a shift of 1 would produce the following substitution table:

```
Plaintext alphabet:     ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ciphertext alphabet:    BCDEFGHIJKLMNOPQRSTUVWXYZA
```

This means the message "HELLO" would be enciphered as "IFMMP".

A Caesar Cipher with a shift of 2 would produce the substitution table

```
Plaintext alphabet:     ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ciphertext alphabet:    CDEFGHIJKLMNOPQRSTUVWXYZAB
```

So "HELLO" would be enciphered as "JGNNQ"

For this problem you are asked to implement a Caesar Cipher in the file **CaesarCipher.java**. In order to do this you will write two methods that have the following "signature":

```
public static String caesarEncipher (String input, int shift)
```

```
public static String caesarDecipher (String input, int shift)
```

These methods will allow you to process the input file on a line by line basis. In addition to these methods you must write an appropriate main program that will adequately demonstrate your cipher. An example run of your program should look like this:

```
Welcome to CaesarCipher

Enter 1 to encipher, or 2 to decipher (-1 to exit): 1
What shift should I use? 2
What is the input file name? hello.txt

What is the output file name? encodedHello.txt
DONE!

Enter 1 to encipher, or 2 to decipher (-1 to exit): 2
What shift should I use? 2
What is the input file name? encodedHello.txt
What is the output file name? originalHello.txt
DONE!

Enter 1 to encipher, or 2 to decipher (-1 to exit): -1
```

Presumably, the file **originalHello.txt** now will contain precisely the original text that is still in the file **hello.txt**. While you are debugging this program, it is probably a good idea to have the content of the input file get "echoed" on to the screen, and also to have the content of the output file also get "echoed." Your methods should *ignore* any character that is not an uppercase alphabetic; in other words, your method should simply copy over unchanged any character that is not upper-case.

You may wish to consider using a **StringBuilder** object for ease in encoding/decoding the characters. To clarify once more: **caesarEncipher()** and **caesarDecipher()** both take one line as their *String* input, and each will return the enciphered/deciphered line. The method that calls **caesarEncipher()** and **caesarDecipher()** is responsible for reading and writing the lines from/to the input and output files.

IV. Supplementary Problems (20 points total)

At most one of the following problems may be answered for some "extra credit."

[8] 20 points use file *Pairs.java*

Write a program that reads the file **`thousand.wrd`** (in the same *unit5* folder that was referred to in problem 6) and prints an alphabetical list of all pairs of letters that do not occur together in any of the words. List each pair only once; for example, if "**qx**" is such a pair, do not list "**xq**" also. Try to make all the pairs fit on the screen when you do the output: the pairs that begin with the letter **a** should appear on one line; the pairs that begin with the letter **b** on the next line, and so on.

Think about using a 26 by 26 two-dimensional array of ***bool***ean values. For example, if the array were named **pairSeen**, then on reading the word '**the**', you can set **`pairSeen['t']['h']`**, **`pairSeen['h']['t']`**, **`pairSeen['h']['e']`**, and **`pairSeen['e']['h']`** all to the value *true*.

Of course, you can't really index the array using the characters 'h' and 't' and 'e' directly. You'll have to devise some other scheme for this.

[9] 20 points use file *Find.java*

Write a program named `Find.java` that searches all files specified on the command line (and prints all lines in these files) that contain a special "word." For example,

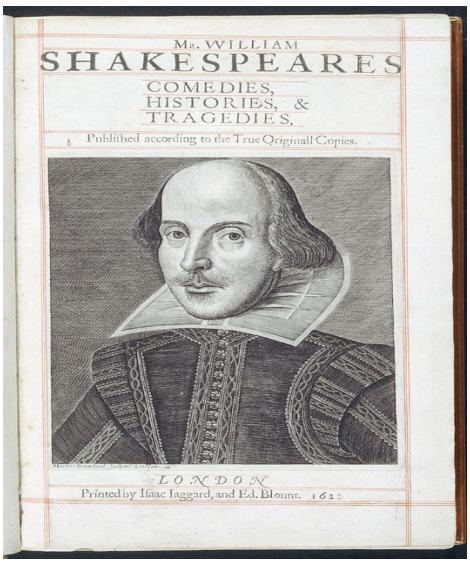
```
java Find Foobar report.txt address.txt Homework.java
```

might result in the following output:

```
report.txt:    an international ring of Foobar bootleggers
address.txt:   Foobar Leitner, Cambridge MA
address.txt:   Foobar Industries, North Pole
Homework.java: String filename = "Foobar.fum";
```

Note that the special word is always the first command line argument.

[10] 20 points



use file *Authorship.java*

Incidents of disputed authorship have not been uncommon. Speculation has existed for several hundred years that some of Shakespeare's works were written by Sir Francis Bacon. And whether it was Alexander Hamilton or James Madison who wrote certain of the Federalist papers is still an open question. Efforts to unravel these "yes [s]he did — no [s]he didn't" controversies often rely heavily on literary and historical clues. But not always. Studies have shown that authors, like burglars, can leave behind "fingerprints."

A given author will use roughly use the same proportion of, say, four-letter words in something she writes this year as she did in whatever she wrote last year. The same holds true for words of any length. BUT, the proportion of four-letter words that Author A consistently uses will very likely be different than the proportion of four-letter words that Author B uses. Theoretically, then, authorship controversies can sometimes be resolved by computing the proportion of 1-letter, 2-letter, 3-letter, ..., 13-letter words in the writing and then comparing it with the same statistics from known authors.

Your task is to write a Java program that computes the above statistics from any text file. Here's what it might look like in action:

```
Name of input file:  RomeoAndJuliet.txt
Proportion of 1-letter words:  3.9%  (74 words)
Proportion of 2-letter words: 18.5%  (349 words)
Proportion of 3-letter words: 24.2%  (456 words)
Proportion of 4-letter words: 19.8%  (374 words)
Proportion of 5-letter words: 11.3%  (212 words)
   …
```

```
   …
Proportion of 11-letter words: 0.7%  (13 words)
Proportion of 12-letter words: 0.4%  (8 words)
Proportion of 13- (or more) letter words: 0.5%  (9 words)
```

[11] 20 points **use file *NoDuplicates.java***

Write a program that reads a text file of numbers of type *int* and outputs all the numbers to another file, but without any duplicate numbers.

Assume the input file is already *sorted* from smallest first, to largest last. After the program is run, the new file will contain all the numbers in the original file, *but no number will appear more than once in the output file*. The numbers in the output file should also be sorted from smallest to largest.

Your program should obtain both file names from the "command line." Be sure to print both the input and output files you tested your program on.

Here is how your program output might appear when it is executed:

```
java NoDuplicates input.txt output.txt

ORIGINAL FILE: input.txt contains the values
3
4
4
4
7
11
11

OUTPUT FILE: output.txt contains the values
3
4
7
11
```