Pset 3 Notes (CSCI E-10B)

General Information

a. Problem set 3 is due Monday, Oct 23, 2023 at 9:00 AM Eastern Time. You can resubmit your assignment as often as desired, but each submission's zip must include every file that you want to be graded.

b. We will deduct 10% for a homework assignment that is turned in up to 3 days late. 20% will be deducted if the homework is more than 3 days late. No homework will be accepted more than 7 days late. The last submission controls the late penalty for the entire assignment.

c. Don't procrastinate! Note, though, that the pset covers material from lecture 7 so you will probably need to defer some questions until you've viewed future lectures and attended or viewed future sections.

d. Your Java code must compile. Programs that do not compile will automatically have their score divided by two.

e. Your programs must behave as specified. Do everything the specs say to do. Do not do more than the specs say to do. Precisely follow the directions in the pset (e.g. use the exact filenames specified in the pset), except make adjustments specified by the staff in these notes or on the Ed discussion board.

f. Some of the work is designated as "extra credit." In this course, extra credit points are kept separate from regular credit points. They only come into play at the end of the semester when Dr. Leitner is assigning final grades. If you are on the cusp between, say, a B+ and an A-, extra credit points can influence that decision.[1]

   If the pset says "identify which problem(s) you want treated as extra credit," ignore that! We automatically allocate points to maximize your regular credit score because that contributes the most to your grade in the course.

   If your submission gets a late penalty, then you won't get any extra credit points for that submission.

g. This course emphasizes programming style. See the Ed posts titled *Reasonably-commented programs*, *Javadoc commenting* and *Java Style Guide*. Inadequately-commented or -styled programs will lose points.

Pencil-and-Paper Exercises

The Simple Exercises must be submitted as plain text files. Plain text files can be created in Visual Studio Code, or by using a programmer's editor like Notepad++, Sublime, TextPad, or by using a plain text editor such as Windows Notepad. Answers submitted in another file format such as **.doc**, **.pages**, **.rtf**, **.pdf**, ... will not be accepted.[2]

Nobody should lose points on the Pencil-and-Paper Exercises programming questions! After you have thought through the problem and written down your answer, check your work by writing simple programs to verify your answer. Don't submit your test programs; only submit your answer for each problem.

a. Problem 1 (**`Prob1.java`**) must be a program class named **`Prob1`**, that prints its command line arguments, one per line, in reverse order.

b. Problem 2 (**`Prob2.txt`**) First, describe a specific circumstance that would cause a method to throw **`NullPointerException`**.[3]

---

[1] The decision is also influenced by your teaching assistant's feedback, so it's a good idea to get to know that person.

[2] Fyi, note that **`.java`** files are stored as plain text files.

[3] Don't say "a program will throw a **`NullPointerException`** if it contains the statement **`throw new NullPointerException()`**" ☺

Then explain why **`NullPointerException`** is NOT a "checked" exception.

Remember, a "checked" exception is an exception that the compiler insists your code "check for" (catch or otherwise explicitly acknowledge). For instance, **`FileNotFoundException`** is a checked exception, so if your program calls a method that might throw a **`FileNotFoundException`**, then your program is required to call that method inside of a **`try/catch`** block that catches that exception, or to otherwise acknowledge that you know that that exception might be thrown.

c. Problem 3 (**`Prob3.txt`**) Explain why the loop doesn't terminate, and provide the actual, specific code for a slightly-revised **`for`** statement that makes the loop work correctly.

d. Problem 4 (**`Prob4.java`**) must be a program class named **`Prob4`** whose **`main()`** method includes the modified code snippet that satisfies the problem's requirements. If desired you can add additional variables to the snippet.

IMPORTANT: Your code must not include any **`throw`** statements. Java will automatically throw the relevant exceptions when appropriate.

e. Problem 5 (**`ExcExample.java`**) This is an extra credit problem. Arrange the problem's "magnet" pieces to produce a program class named **`ExcExample`** that works as specified. You cannot edit any of the pieces. You cannot divide a piece's code into more than one piece. You cannot add anything other than individual curly braces. You must use all of the pieces.

Programming Problems

Submit complete and correct programs that behave exactly as specified, except make adjustments specified by the course's staff in these notes or on the Ed discussion board.

Your programs' interaction with the user should match what's in the assignment, and when sample cases are shown in the problem set, your program must produce the same output when given the sample case's input.

Whenever you are asked to write a specific method that's not **`main()`**, calling that method must not cause any output to be sent to the terminal unless the instructions say otherwise.

Unless otherwise specified, programs that read or write files must behave gracefully if input files cannot be found, cannot be read, are empty, contain unexpected data, ... or if output files cannot be written.

Programs that accept command line arguments must behave gracefully if an unexpected number of command line arguments are provided.

If you are told to provide code that demonstrates your method, you must follow these rules:

1. You must not require your TA to define the test cases.
2. You must not require your TA to read your source code in order to decipher your code's output.

Instead, you need to implement a robust set of self-documenting test cases that convincingly demonstrate your method. Here's how I would do that: For a well-chosen set of test cases, my demo would contain statements analogous to:

```
System.out.printf( "someMethod( %d, %d ) = %d\n", int1, int2,
                                                   someMethod( int1, int2 ) );
```

You will replace "**`someMethod`**" with the actual name of the method being demonstrated, and you'll adjust the number of arguments, their data types, the method's return type, etc.

a. Problem 6 (**`ExamAnalysis.java`**) must be a program class that:
   1. Reads a line from the user's keyboard that contains the correct answers to the exam questions.
   2. Reads a line from the user's keyboard that contains the path to the file with the students' responses.
   3. Reads the students' responses from that file, analyzing them and producing the specified output.

   Your program's output should match what's shown in the pset when you test with the **`exams.dat`** response file from the Java Resources page and the answer key **`ABCEDBACED`**.

   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. What's the best way specify the maximum supported number of exam questions and the maximum supported number of students? Hint: it should be trivial to change those numbers. Hint: **DO NOT ASK THE USER FOR THOSE VALUES**.

   You might find **`ArrayList`**s to be useful for this program. That data structure will be discussed in lecture 7.

   This is a relatively complicated program, so it will benefit from stepwise refinement. Don't put a lot of code in **`main()`**. Instead, decompose the program into several well-focused methods. Work on it a piece at a time. E.g.:
   1. Have **`main()`** call a method that gets a line from the user's keyboard that contains the correct answers to the exam questions. After you get that to work ...
   2. Then have **`main()`** also call a method that reads the path to the responses file and gets the responses. After you get that to work ...
   3. Then have **`main()`** also call a method that does the student analysis, outputting the count of the correct, incorrect and blank responses for each student. After you get that to work ...
   4. Then have **`main()`** call a method that does the question analysis.

   After you get the program "working," take deep breath and go for a walk. When you return, take a critical look at your code. How might it better be organized? Are the comments lucid? Are some of the methods still too long or complicated? Are the variable names and method names meaningful? ...

b. Problem 7 (**`CaesarCipher.java`**) This problem is required for graduate students, and is extra credit for undergrad students.

   The program's interaction with the user must look like what's in the pset. The program must read an input file line by line. After an individual line is read, the individual line must be transformed by one of these methods:
   - **`public static String caesarEncipher (String input, int shift)`**
   - **`public static String caesarDecipher (String input, int shift)`**

   The transformed line must then be written to the output file. The read-a-line, transform-the-line, write-the-line cycle repeats until all lines in the input file have been processed.

   Your **`caesarEncipher()`** and **`caesarDecipher()`** methods must ignore any character that is not an uppercase alphabetic. "Ignore" means that the character should be copied as-is to the result String, without having any encipher/decipher processing applied to it.

   As usual, your program must reference **`char`** constants instead of the constants' numeric values.

   Can you think of a way for one of the **`Caesar(en,de)cipher()`** methods to leverage the other one so that the body of one of them is just a single line?

Can you construct your algorithm so that there are no restrictions on the value of the shift variable? You're only required to support shift counts between -25 and +25 inclusive, but you might find it interesting to try to support arbitrary shift counts. I suggest that you do this in two steps:

1. Convert large-magnitude shifts to shifts between -25 and +25.
2. Convert negative shifts to positive shifts.

By the way, if you echo the input and output files to the user's screen while you are debugging your program, you must remove that code when you submit your program.

Supplementary Problems

All students can optionally solve one Supplementary problem for extra credit.

The guidelines discussed in the beginning of the **`Programming Problems`** section also apply to this section.

a. Problem 8 (**`Pairs.java`**) Copy **`thousand.wrd`** from the Java Resources page to the same folder that **`Pairs.java`** is in. **`Pairs.java`** reads **`thousand.wrd`** and prints an alphabetical list of all letter pairs that do *not* occur together in any of the words.

   In other words, track all possible letter pairs. For each <*ltr*$_1$, *ltr*$_2$> pair, remember whether that pair or its reverse <*ltr*$_2$, *ltr*$_1$> occur sequentially in some of the file's words.

   Now that you know which pairs **do** occur sequentially in the file's words, you also know which pairs **don't** occur in the file's words. Display the "don't occur" pairs. List each pair only once; for example, if **`qx`** is such a pair, do not also list **`xq`**. List the pairs that begin with **`'a'`** on one line, the pairs that begin with **`'b'`** on a separate line, etc.

   Solve the problem by using a 26×26 array of **`boolean`** values, and do not use any of Java's sorting methods.

   You might recall from an earlier lecture that the regexp **`[^\w']+`** can be used with the Scanner class to retrieve non-**`[a-zA-Z0-9_']`**-delimited words. Now consider the word **`don't`**. In my opinion it has three pairs of letters (**`do`**, **`on`**, **`nt`**). What's your opinion? Be sure that you document that opinion in a comment in your program.

b. Problem 9 (**`Find.java`**) Write a program that accepts a word as the first command line argument, and accepts any number of files as the subsequent command line arguments. For each file, print every line that contains the word. Your output should look like what's in the assignment.

c. Problem 10 (**`Authorship.java`**) The program asks the user to specify a file, and then it analyzes that file to produce the specified output, formatted as specified. The regexp mentioned above will be useful here too.

   Test your program on the file named **`RomeoAndJuliet.txt`** file located at the Java Resources page.

d. Problem 11 (**`NoDuplicates.java`**) The program accepts two command line arguments: the first argument specifies an input file and the second argument specifies an output file.

   Assume that the input file contains zero or more **`int`**s per line, and that the input file's **`int`**s are sorted.

   Copy the input file's **`int`**s to the output file, preserving order, one **`int`** per line, except only copy one occurrence of each **`int`**.

   If the input file has zero **`int`**s, you should produce an empty output file. If the input file does not exist, you should not produce an output file.

You must write the output file as the input file is being read, instead of first reading the entire input file into a data structure.

This problem is similar to the **`Increase.java`** program from last semesters' pset 3.