Pset 4 Notes (CSCI E-10B)

General Information

a. Problem set 4 is due Monday, Nov 13, 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 lectures 8, 9 and 10 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

This assignment has no pencil-and-paper exercises.

Programming Problems

To minimize complications when developing Swing programs with VS Code, you are advised to run VS Code on your local PC instead of running the cloud-hosted VS Code.

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.

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

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:

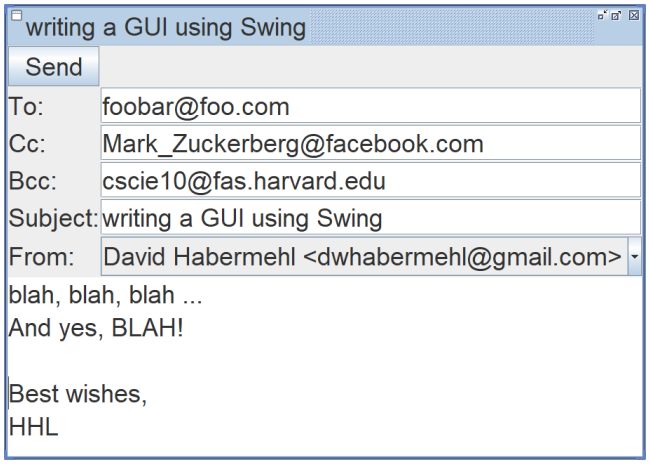
```
System.out.printf( "someMethod( %d, %d ) = %d\n", int1, int2,
                                                   someMethod( int1, int2 ) );
```

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 1 (**Age.java**) is easy. All you need to do is
   1. Display this popup using a **JOptionPane** method:
   2. Retrieve the user's response
   3. If the user types a number less than 40, respond with a message box saying that he or she is young. Otherwise, respond with a message box that teases the user for being old.

   For this problem don't use any Swing classes besides **JOptionPane**.

b. Problem 2 (**MailLayout.java**) requires you to present a **JFrame** laid out to resemble an email program's "new message" interface:
   1. The boxes next to the To:, Cc:, Bcc: and Subject: fields are all **JTextFields**.
   2. The box to the right of From: is a **JComboBox**. You should include 3 different names/email addresses in this dropdown list.
   3. The box that contains the email message's body (blah, blah, blah … thru HHL) is a **JTextArea**.
   4. The **JFrame**'s "title" should be set to whatever the user types inside the Subject: **JTextField**. Whenever the Subject field is blank, the title should simply be "New Message". One way to accomplish this is to add a focus listener to the Subject: **JTextField**.
   5. Include a clickable "Send" **JButton** located just below the title bar of the **JFrame** or anywhere else that makes sense. When clicked, this button should cause the message's body to be written to a file named **outbox.txt**. After the message's body is written to the file, the fields' contents should be cleared and the title should be reset.



One of the challenges in this problem is to create a layout that resizes gracefully:

1. When the **`JFrame`** is resized horizontally:
   a. **`JLabel`** widths should not change.
   b. Send button width should not change.
   c. Other components' widths should change.
2. When the **`JFrame`** is resized vertically:
   a. **`JLabel`** heights should not change.
   b. **`JTextField`** and **`JComboBox`** heights should not change.
   c. Send button height should not change.
   d. Other components' heights should change.

One way to create a layout that resizes gracefully is to use nested **`JPanel`**s, where the **`JPanel`**s use the layout manager best suited to manage that particular **`JPanel`**'s look and feel.

<u>You can **either** solve problem 3, **or** you can solve **both** problem 4 and problem 5</u>.

c. Problem 3 (**`TrafficLight.java`**) : Populate a **`JFrame`** with widgets and graphics that resemble a traffic light. For up to three extra credit points, implement a way to interact with the display to dim and brighten combinations of the lights. For example, clicking a light could brighten it while simultaneously dimming the other two lights.

   A common mistake is to truncate the red, green and yellow circles. Make sure that your program displays complete, untruncated round circles.

d. Problem 4 (**`BullsEye.java`**): Populate a **`JFrame`** with widgets and graphics that resemble a target: alternating black and white concentric rings as shown in the pset.

   The easiest way to do that is to draw a filled black circle, then draw a smaller filled white circle on top of the black circle, … .

e. Problem 5 (**`Currency.java`**): Write a program to convert currency between euros and U.S. dollars. Provide two **`JTextField`**s for the euro and dollar amounts. Between them, place two **`JButton`**s labeled **`>`** and **`<`** for updating the euro or dollar amounts. For this exercise, use a conversion rate of 1 euro = 1.13 U.S. dollars. Be sure to use **`JLabel`**s to inform the user how to use the program.

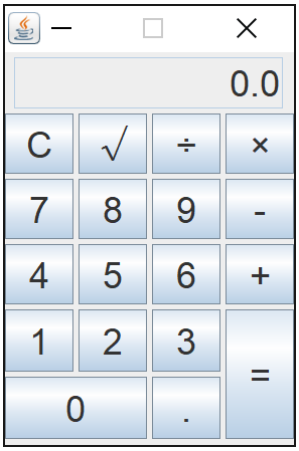
<u>Problem 6 is required for graduate students! Other students may solve it for extra credit</u>.

f. Problem 6 (**`FifteenPuzzle.java`**): Write a Java application that presents a 15-puzzle. If the user clicks on a number tile that is horizontally- or vertically-adjacent to the blank tile, then the clicked tile is swapped with the blank tile. Clicks on other tiles should be ignored. The goal is for the player to sort the tiles numerically.

   <u>The program must present a solvable puzzle</u>. Initially, display the tiles in numeric order, with the blank tile in the lower-right corner. When the user clicks a "Shuffle" **`JButton`**, programmatically execute some number of random legal moves. A move is legal if it's a click on a number tile that is horizontally- or vertically-adjacent to the blank tile.

   The more shuffle moves you programmatically execute, the harder the puzzle is to solve. Give the user control over that parameter, maybe via a **`final`** variable, or maybe via a **`JTextField`**, or maybe by executing one random, legal move every time the "Shuffle" **`JButton`** is clicked.

g. Problem 7 (**`Calculator.java`**, **`CalcBackend.java`**, **`CalcBackendTest.java`**): Implement a calculator (similar to the one that's built-in to both the Macintosh and Windows OSes). Your program should look something like this when it starts up:



This is probably the most challenging program you will encounter this semester.

You **must** implement this program in two separate classes:

1. **`Calculator.java`** manages the GUI.
2. **`CalcBackend.java`** keeps track of what the GUI should display based on the user's button clicks.

High-level flow: When the user clicks one of the display's **`JButton`**s:

1. In **`Calculator.java`**, the **`JButton`**'s action listener calls a **`CalcBackend`** instance method (**`feedChar`**) to tell **`CalcBackend`** to update the calculator's state in response to the clicked button.
2. Then the **`JButton`**'s action listener immediately calls another **`CalcBackend`** instance method (**`getDisplayVal`**) to get the new **`String`** that it will use to populate the display's **`JTextfield`**.

Here are more details on the **`Calculator`** and **`CalcBackend`** classes.

1. **`Calculator.java`**:
   a. Creates a **`CalcBackend`** object
   b. Lays out the **`JFrame`**
   c. Adds action listeners for the **`JButton`**s
2. **`CalcBackend.java`**: keeps track of what should be displayed on the screen by updating the calculator's state based on button clicks that it's told about from **`Calculator.java`**'s **`JButton`**s' action listeners.

```
// CalcBackend must implement these public methods. It cannot
// expose any other non-private methods or variables, and it
// cannot access anything in the Calculator class.
public class CalcBackend {

    // Constructor
    public CalcBackend() {
        //Initialize the calculator's internal state
    }

    // Calculator.java calls feedChar to pass button clicks to CalcBackend
    public void feedChar( char c ) {
        // Update calculator's internal state per clicked button
    }

    // Calculator.java calls getDisplayVal to see what the calculator should display
    public String getDisplayVal() {
        // Return the new String for the GUI to display.
    }

    // Other private helper methods as required.
}
```

Important note #1: The **only** communication and interaction between the **`Calculator`** class and the **`CalcBackend`** class is via the **`CalcBackend`** class's **`feedChar()`** and **`getDisplayVal()`** instance methods. If you use **any** other means besides those two methods for the two classes to communicate or interact with each other, a significant point deduction will be taken.

Important note #2: I will test your **`CalcBackend`** class by feeding it button clicks via **`feedChar`** and then immediately calling **`getDisplayVal`**. If the **`String`** returned by **`getDisplayVal`** does not faithfully represent the result of the button clicks, then that test will fail. Since the **`CalcBackend`** tests are completely independent of **`Calculator.java`**, be sure that **`Calculator.java`** has no involvement in constructing the **`String`** to be displayed.

`CalcBackend` should do all of its numeric computations using the **`double`** data type. Here's an easy way for **`getDisplayVal`** to convert a **`double`** value into a **`String`**:

```
// Assume CalcBackend maintains displayVal to contain current value to display.
private double displayVal;
...
public String getDisplayVal() {
    String displayString = "" + this.displayVal;
    ...
    // Adjust displayString as necessary, say to show multiple
    // trailing zeroes to the right of the decimal point, or to
    // limit the length of displayString.
    ...
    return displayString;
}
```

One reason this problem is particularly challenging is that the program must produce repeatable and reasonable results for any sequence of button clicks. One way to manage this level of complexity is to create a list of "states" that represent what the calculator is currently doing. For instance:

1. Ready for first operand
2. Ready for second operand
3. Constructing operand (left of decimal point)
4. Constructing operand (right of decimal point)

Then you might create categories of buttons. For instance:

1. Digit
2. Decimal Point
3. Equals Sign
4. Binary Operator (+ - ÷ ×)
5. Unary Operator (√)

Now when **`feedChar`** runs, it can say "the calculator is in state **x** and the clicked button is in category **y**." In the above scenario, there are just 4 states × 5 button categories = 20 possible combinations.[2] Any button click, in any sequence of clicks, necessarily fits into one of those combinations and can have a dedicated method to handle that specific combination.

[2] You might well decide that a different set of states and/or a different set of button categories is necessary.

For instance:

```
// state = constructingOperandLeftOfDecimalPoint
// button category = DecimalPoint
private void constructingOperandLeftOfDecimalPoint_DecimalPoint {
    this.state = constructingOperandRightOfDecimalPoint
}

...
// state = constructingOperandRightOfDecimalPoint
// button category = DecimalPoint
private void constructingOperandRightOfDecimalPoint_DecimalPoint {
    // Do nothing; ignore the bogus decimal point click.
}
...
```

Extra Credit Problems

All students can optionally solve as many extra credit problems as desired.

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

a. Problem 8 (**`Clock.java`**): Write a Java application using Swing graphics that draws the face of an analog clock with an hours hand and a minutes hand, showing the time that the user enters in two **`JTextField`**s (one for the hours, one for the minutes). The pset suggests (and I concur) that you skip this problem if you don't remember your trigonometry.

b. Problem 9 (**`Olympics.java`**): Draw the specified figure, but don't worry about the complex layering that you see in the pset. Create a method named **`drawRing`** that draws a ring at a given position and color (supplied as arguments to the method).