CS 235 - Homework 6

Deadline:

11:59 pm on **Friday, October 22, 2021.**

Purpose

To check your understanding of, and to practice using, Java's `FlowLayout`, `BorderLayout`, and `GridLayout` layout managers.

How to submit:

You complete **Problem 1** on the course Canvas site (short-answer questions related to Java's basic layout managers).

For **Problems 2 onward**, you will create the specified `.java` files, and then submit those to the course Canvas site. (You'll be creating `.class` files, also, but you do not submit those.)

Important notes:

- Note that Java applications with graphical user interfaces are expected to be structured as demonstrated in the in-class example `ButtonTest.java`
  - (that is, with an application class that creates and displays a `JFrame` subclass instance within the event dispatch thread,
  - and a `JFrame` subclass that creates and adds a `JPanel` subclass instance to itself in its constructor, and
  - a `JPanel` subclass whose constructor creates and adds appropriate components to itself)
- Because graphical user interfaces are involved, **the CS50 IDE will NOT work on this homework's problems**. (Running in a browser, on the cloud, it cannot access your screen to display a `JFrame`.)

  If you have a Terminal or bash shell, you can compile and run Java as you do from the CS50 IDE Terminal.

  AND -- I have verified that Java works -- compiles and runs -- from the **Command Prompt** on `vlab.humboldt.edu` as it does from the CS50 IDE, also.

  That is:
  - Log into `vlab.humboldt.edu`
  - In the search bar on the lower left, search for "command prompt", and click on the "Command Prompt" app that comes up.
  - Even though this is a Windows Command Prompt window and not a bash shell, commands such as `mkdir` and `cd` and `ls` work here.
  - I found that if I saved a `.java` file on the vlab desktop, then from the Command Prompt I could do

the following:

```
C:\Users\st10> cd Desktop
C:\Users\st10\Desktop> javac MyGuiApp.java
C:\Users\st10\Desktop> java MyGuiApp
```

...and my application would compile and run!

- (But save your `.java` files to your Google Drive for safer, longer-term storage that can be more easily accessed than the vlab Desktop!)

- Follow the class Java coding standards mentioned in class and demonstrated in posted in-class examples -- some of these include:
  - Follow the Java naming standards that have been discussed in class.
  - Attempt "javadoc-style" comments for **each** Java class and method, in the same style as you see in posted in-class examples.
  - Everything inside a set of { } must be indented by AT LEAST 3 SPACES -- and the beginnings of statements that are sequential should be indented the SAME NUMBER of spaces. (That is, sequential statements should line up.)
  - { and } should each go on their OWN line, with { lined up evenly with the preceding line, with the { }'s contents indented by at least 3 spaces, and with } lined up with the opening {. That is, handle the curly braces as you see in all posted class examples!
- ASK ME if any of these are unclear to you!

Problem 1 - 21 points

Problem 1 is correctly answering the "HW 6 - Problem 1 - short-answer questions on basic layout managers" on the course Canvas site.

Problem 2

Consider `ColorPlay1.java` from Homework 5 - Problem 3 – remember that there is also a posted example solution, available from the course Canvas site, under " Selected solutions - Homeworks...", if you prefer.

Modify either your Homework 5 solution or the posted solution into a file `ColorPlay2.java`, whose modified classes meet the following specifications:

- `ColorPlay2Panel` should now use **`BorderLayout`**, and in its **northern** region should be a **sub-panel** using **`FlowLayout`** whose contents include the `JLabel` including your name (although it may also include more that just that `JLabel`).
- `ColorPlay2Panel`'s **center** region should somehow include a **sub-panel** using **`GridLayout`**, with the grid arranged as you choose but with **at least 6 cells**, containing the red-green-blue labels and textfields.
  - (That is, `ColorPlay2Panel`'s **center** region might be filled with this sub-panel using `GridLayout`, or it might contain a sub-panel using, for example, `BorderLayout` whose center

region happens to include the required sub-panel using `GridLayout`...)

- `ColorPlay2Panel`'s **south** region should contain a **sub-panel** (a button-panel) that uses **`FlowLayout`** whose contents include the button that is pressed to set the background color to the currently-entered red, green, and blue values.
- Decide which panel's or panels' background color will be set to the specified RGB values -- as long as it is clear and obvious that something has been set to a color based on the current red, green, and blue values when the button is clicked, that should meet the specifications.
  - (But please ask me if you have any concerns about this requirement, or any of the others for that matter!)
- (If you would like to put anything in the east and west regions -- such as "dummy" labels of blanks for spacing -- that is fine.)
- Colors (besides that being set by the user-specified red, green, and blue values) and font-sizes and font-styles are up to you, as long as the result is readable and attractive.

Submit your resulting `ColorPlay2.java`.

Problem 3

Copy `GameDie.java` into your directory where you are putting this problem's Java files. You will be using an array of `GameDie` instances in this problem. (This can be the Week 1 Lab's version of `GameDie`, or your Week 4 Lab Exercise version of `GameDie`.)

You made a GUI application, `DieRoller`, that rolled a single die in Homework 4 - Problem 3. But, now that we have `GridLayout` and `BorderLayout` available, writing a version that allows you to roll multiple dice should now be more reasonable (from a layout point of view).

Decide: would you like to roll 2 dice? 3? 4? 5? Choose, make this size a named constant, and create a `DiceRoller` application that uses an array of `GameDie` of that size, and also meets the following specifications:

- Decide if you want to use a named constant for the number of sides per die, or if you want to somehow allow the user to specify this (via, for example, an appropriate `JOptionPane`).
- The north should contain a sub-panel with a centered, descriptive label including your name.
- The center should contain a sub-panel with a two-row grid such that the top row contains buttons to roll their respective die, and the second row contains labels and/or output textfields - your choice - giving the value of the latest roll of that die
  - (Whether there are any sub-panels within this sub-panel is up to you.)
- The south should contain a sub-panel that has, centered, the sum of the *latest* rolls of all of the dice, using a label and/or output textfield, your choice.
  - (Remember, you have an array of `GameDie` instances -- iterating through it and adding all of their current top values should be quite reasonable.)
- Use at least one visible border somewhere in your application
- (If you would like to put anything in the east and west regions -- such as "dummy" labels of blanks for

spacing -- that is fine.)

- Colors and font-sizes and font-styles are up to you, as long as the result is readable and attractive.

Submit your resulting `DiceRoller.java`.

Problem 4

During the Week 7 Lecture, it was briefly mentioned that, within an `actionPerformed` method, you could grab the text associated with the button that was clicked using the `actionPerformed` method's `ActionEvent` parameter's `getActionCommand` method. And, this is demonstrated in the posted `LayoutTrio1.java` example.

But, you can grab more than the action command of the action event -- you also can actually grab a reference to the SOURCE of an event by using that `ActionEvent` parameter.

The `ActionEvent` class includes a method `getSource`, which returns a value of type `Object`, a reference to the component that was acted upon (here, a reference to the button that was clicked).

But what if you want to call `JButton` methods on the `Object` returned? Well, if you know the `Object` is also of type `JButton`, you can cast it to `JButton`.

That is, in `LayoutTrio1.java`, in inner class `NumButtonAction`, I could rewrite its `actionPerformed` method using this as follows:

```
public void actionPerformed(ActionEvent event)
{
    // grabbing the SOURCE of this event -- here, a button!
    //     (getSource() returns an Object -- I *know* it
    //     is a JButton, here, so it is safe for me to
    //     cast this returned result to a JButton)

    JButton clickedButton = (JButton) event.getSource();

    int currButtonVal =
        Integer.parseInt(clickedButton.getText());

    runningTotal += currButtonVal;
    resultsField.setText( Integer.toString(runningTotal) );
}
```

And -- as you can see above, you can get the text on the top of a `JButton` using its `getText` method.

Interestingly enough, you can also CHANGE the text on a `JButton` using its `setText` method! So, with that noted, on to this problem!

Create a simple tic-tac-toe game **BOARD** program, whose classes are in a file `TicTac.java`. Since this problem is more about layout practice than about game logic, please note that **it is NOT required to be very "smart"!** It **JUST** needs to meet the **following** specifications:

- In the north, it needs to have a label stating "`Tic-Tac-Toe - by` <your name here>" (You can choose if you want this label on a sub-panel or not.)
- In the south, it needs to have a `JPanel` using `FlowLayout`, containing a "Clear" button. (So, this

button will be centered within a South sub-panel, rather than taking up the entire panel)

- In the center, it needs a `JPanel` containing a 3x3 grid of buttons using `GridLayout`. Let's call these the game buttons.
- When the application starts, all of the game buttons are **blank**.
  - If you click on a blank game button, it should change to show a large X. If you click on a game button with an X, it should change to show a large O. If you click on a game button with an O, it should change to be blank.
- If you click on the Clear button, all of the game buttons should change to be blank.
- Do NOT use a 9-way `if-else-if` (or 9 `if` statements, either...!) to handle the 9 buttons within your code!!
  - Use an **array of `JButton` objects!**!
- Use at least one visible border somewhere in your application
- (If you would like to put anything in the east and west regions -- such as "dummy" labels of blanks for spacing -- that is fine.)
- Colors and font-sizes and font-styles are up to you, as long as the result is readable and attractive. Make sure the X's and O's on the buttons are large and easy to see!

OPTIONALLY:

- **IF** you'd like, you can add more sophisticated logic to the above, IF you still meet the above requirements AND CLEARLY DOCUMENT those logic enhancements.

Submit your resulting `TicTac.java`.