CSC 171 LAB 09

Lab Instructions

The goal of this lab is to give you practice with static and non-static class members, the Singleton design pattern, null references, graphics, and GUIs.

For this lab, implement all of the instructions below in one or more appropriately named packages. For the exercises in this lab it is acceptable to implement your "test" as a main method within the same class.

1. Loggers are often used to log important messages to the console or to log files, including errors and other events. Implement a class called "**`Logger`**" that has a non-**`static`** method called "**`log`**". This method should take a **`String`** as a parameter and print the string.

2. The Singleton design pattern guarantees that there will be exactly one instance of a class available per Java process. Loggers are often singletons so that one object is responsible for receiving and logging messages to the appropriate place. This is especially important for file-based loggers as only one object may write to a file at a time, so multiple loggers would get in each other's way. Convert your **`Logger`** class to a singleton by following these instructions:
    a. Create a **`private`**, parameterless constructor for the **`Logger`** class. This will prevent any other classes from using the **`new`** operator to create an instance of your **`Logger`**.
    b. Create a **`private static`** variable of the type **`Logger`** in the **`Logger`** class and assign it a **`null`** value.
    c. Implement a **`public static`** method called "**`getLogger`**" that returns the **`static`** instance variable that you created. If (and only if) your **`Logger`** is **`null`**, this method should create a new **`Logger`** and assign it to the class variable before returning it.
    d. Recall that **`static`** methods can be called *without* an instance of the class simply by using the class name before the dot operator and the method name, e.g. **`Math.pow( a, b )`**. Verify that calling your **`getLogger`** method two or more times returns *exactly the same* **`Logger`**.

3. Create a class called "**`Equalizer`**" that has a single method called "**`isEqual`**" that takes two object parameters. **`Equalizer`** should return "true" if any of the following are true, and false otherwise:
    a. Both objects are null
    b. The "**`==`**" operator returns true.

c. The “equals” method on either object returns true when called with the other object as the parameter.

Test your method with the following combinations of values:

a. two **`null`** references (**`null`**, **`null`**)
b. one null and one non-**`null`** reference (e.g. **`null`**, **`"abc"`**);
c. one non-**`null`** reference and one `null` reference (e.g. **`"abc"`**, **`null`**)
d. the same non-**`null`** object twice (e.g. **`"abc"`**, **`"abc"`**)
e. two different objects that are equal (e.g. new Integer(123), new Integer(123))
f. two different objects that are not equal (e.g. **`"abc"`**, **`"def"`**)

4. Create a class called “**`PanelFrame`**” that extends **`JFrame`**. In the constructor of your **`PanelFrame`**, do the following:
   a. Create a new **`JPanel`**
   b. Set the background color of the **`JPanel`** to the color of your choice (as long as it’s not gray).
   c. Use the **`add`** method to add the **`JPanel`** to your **`PanelFrame`**.
   d. Set the size of the **`PanelFrame`** to at least 500x500 pixels.
   e. Set the default close operation to exit on close.

   Write a **`main`** method that creates a new **`PanelFrame`** and sets it to be visible.

5. Implement a class called “**`Grid`**” that extends **JFrame**. Create a constructor that takes two integers as parameters: one to indicate a number of rows, and one to indicate a number of columns. Recall that the “**`getSize`**” method on any component will return a **`java.awt.Dimension`** object that contains the width and height of the component in pixels. Override the **`paint`** method and use **`getSize`** to divide the **`Grid`** into the appropriate number of rows and columns by drawing horizontal and vertical lines. You may use the background and foreground colors of your choice provided that the lines are clearly visible. Your grid should respond appropriately when the size of the frame is changed with the mouse (but do not worry about handling very small frames).

6. Implement a class called “**`Checkerboard`**” that extends **`JPanel`** and draws an 8x8 checkerboard of alternating red and black squares that is at least 500x500 pixels. When setting your colors, **do not** used the constants defined in the **`java.awt.Color`** class. instead, construct the **`Color`** object using the appropriate RBG values. Display your **`Checkerboard`** in a **`JFrame`**. Your checkerboard should respond appropriately when the size of the frame is changed with the mouse (but do not worry about handling very small frames).

7. Implement a class called “**`Bullseye`**” that extends **`JPanel`** and draws a series of concentric ovals. You may choose the color for any oval except for the smallest two, which should be white and red. Your bullseye should respond appropriately when the

size of the frame is changed with the mouse (but do not worry about handling very small frames).

8. Create a GUI that uses a **`GridLayout`** of N rows and M columns. Fill the layout with **`JLabels`**, each of which displays a unique text string, e.g. "Label 0", "Label 1", "Label 2", and so on. Your GUI should work for any (reasonable) values N > 0 and M > 0.

9. Begin by copying the GUI that you created in step 8 into a new class. Modify your code so that each **`JLabel`** is contained within a **`Border`** of your choice.

10. Create a simple GUI of your own design. You must incorporate *at least* 3 of the following:
    a. Use of three different colors
    b. Use of three different components that are **not** containers
    c. A non-default layout manager
    d. An overridden paint method that draws custom shapes/lines/fonts
    e. Borders
    f. A layout manager within a layout manager (e.g. a **`GridLayout`** used on one **`JPanel`** within a **`BorderLayout`**).

HAND IN

Before handing in, create two additional files in your lab directory:

1. Create a README that contains the following:
    a. Your contact information (name, class, lab session), TA name, and assignment number.
    b. A brief (one paragraph at most) description of the assignment.
    c. Instructions explaining how to run your code. While it is possible that some TAs will compile, run, and test your code in Eclipse, it is also possible that they will want to compile and run it from the command line. You should include instructions on how to compile and run each of the executable Java classes.
2. Create a file titled "SampleOutput" showing the results of running your code. (You may copy and paste from the Eclipse console). If appropriate, please include a comment above the output for each section (e.g. "#OUTPUT FOR SECTION 1") and put a few blank lines between each section.

Hand in by uploading the compressed (i.e. "zipped") folder containing your assignment files to Blackboard.