Unit 6 Problem Set

I. Overview

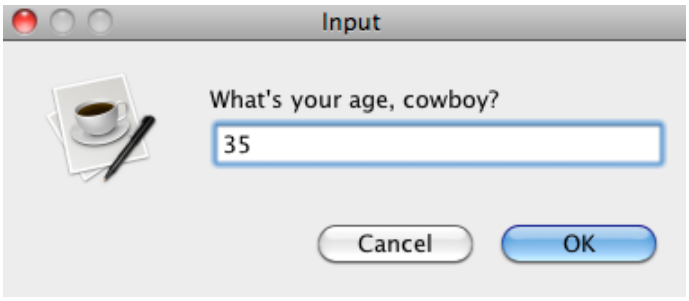
This problem set is due prior to 5 PM on Monday, April 9. The assignment is designed to reinforce your understanding of *Swing* and event-driven programming techniques, as well as other related concepts. Consult the Reges & Stepp textbook (chapter 14) as well as Oracle's on-line documentation at http://docs.oracle.com/javase/7/docs/api/index.html as needed.

Remember that to display a GUI using *Swing* when developing Java applications on *Cloud9*, you must first run an installation script. Detailed instructions for doing so appear on our course website.[1]

II. Programming Problems (110-155 points)

[1] 10 points *use file Age.java*

Write a *Java* program that pops up an *option pane* that asks the user for his or her age, e.g.,

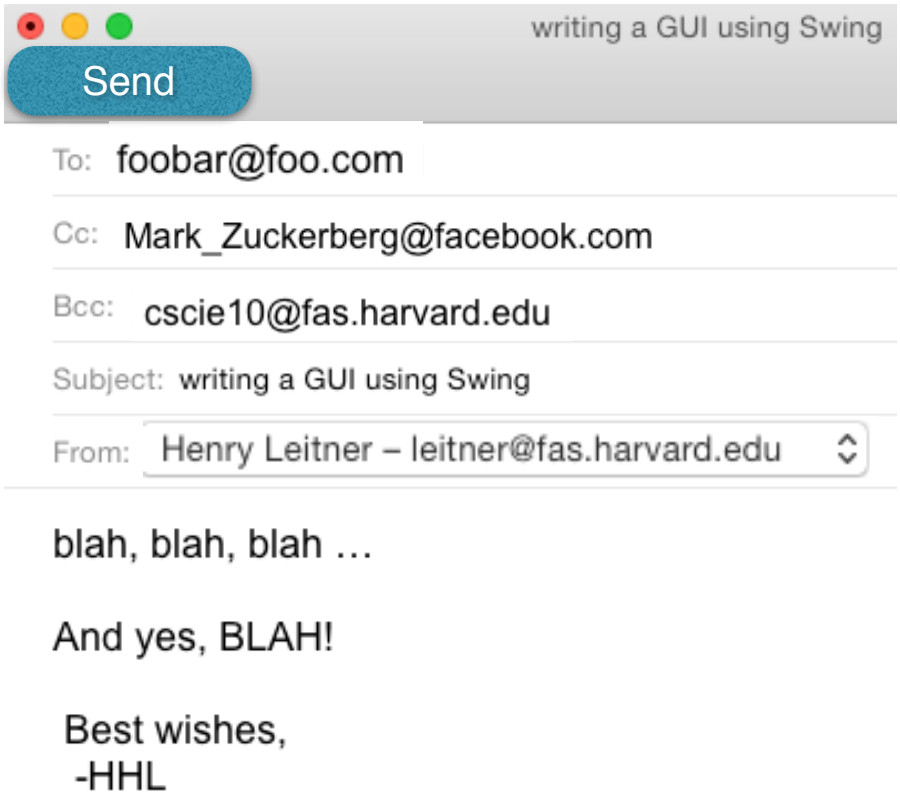


If the user types a number less than **40**, respond with a message box saying that he or she is young. Otherwise, respond with a message box that teases the user for being old!

[1] The alternative would be to run such applications on a personal computer that has Java installed, e.g., using the *Terminal* program under Macintosh OS X.

[2] 25 points use file *MailLayout.java*

My mail program presents a window that looks like this when I want to compose a new message:



Write a complete Java program that presents a window (**JFrame**) organized like this. The boxes next to the fields *To:*, *Cc*:, *Bcc*: and *Subject*: are all **JTextFields**. The box to the right of *From*: is a **JComboBox** (you should include 3 different names/email addresses in this dropdown list).

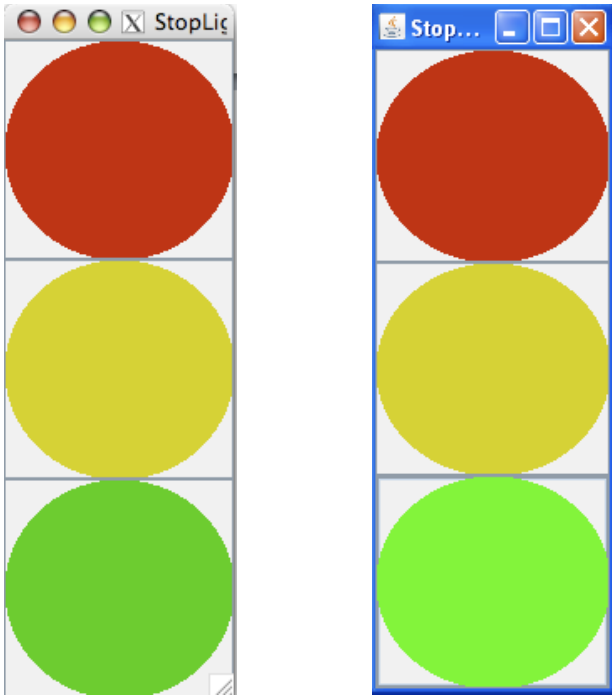
The region that contains my email message (*blah, blah, blah …*) is a **JTextArea**. Also, you should include a clickable "*Send*" button (a **JButton** object). It can be located just below the title bar of the JFrame, or anywhere else that makes sense. When clicked, this button should cause the content of the **JTextArea** to be written to a file named **outbox.txt** After the message is written to file, the screen should be cleared and title should be reset.

Note that the JFrame's "title" should be set to whatever the user types inside the *Subject:* **JTextField**. If the user does not type anything, then the title should simply be *"New Message"*.

Solve either problem [3], or both problems [4] and [5].

[3] 25 points use file *TrafficLight.java*

Write a *Java* application named **TrafficLight** that produces a drawing that resembles a traffic light.[2] Arrange for each circle to have the proper color. Here's what your program might look like in action running on both Mac OS/X and on Windows:



For a small amount of "extra credit" have a control on-screen to dim and to light up individually each of the three lights. For example, clicking the the green light could brighten it a bit, while at the same time "dimming" the other two.

[2] You could utilize such methods as **drawOval**, **fillOval**, and **drawRect**; all three take 4 arguments: **x**, **y**, **width**, and **height**. Or you could use a **JPanel** (sized correctly) and set its' background color. You decide what works best!

[4] 10 points **use file *BullsEye.java***

Write a complete Java program that draws a "bull's eye"—a set of concentric rings in alternating black and white colors, as shown here.

Hint: Fill a black circle, then fill a smaller white circle on top, and so on.

[5] 15 points **use file *Currency.java***

Write a *Swing* application to implement a currency converter between euros and U.S. dollars, and vice versa. Provide two text fields for the euro and dollar amounts. Between them, place two buttons labeled **>** and **<** for updating the field on the right or left. For this exercise, use a conversion rate of 1 euro = 1.23 U.S. dollars.

[6] 25 points **use file *FifteenPuzzle.java***

This problem is required for graduate-credit students only! Undergraduate-credit students may solve the Fifteen Puzzle problem for up to 25 points of "extra credit."

According to Wikipedia, "The **15-puzzle** (also called **Gem Puzzle**, **Boss Puzzle**, **Game of Fifteen**, **Mystic Square** and many others) is a slid-

ing puzzle that consists of a frame of numbered square tiles in random order with one tile missing." Here's an example:

A user clicks on one of the numbered tiles that's adjacent to the empty space, and that tile then moves to the empty space. For example, if I click the 8, the result looks like this:

Write a Java application that presents a 15-puzzle with the numbered tiles in some random order. Your program should allow the user to click on numbered tiles in order to move them around. The actual objective of the puzzle is for the player to get the board to look like this (numbers sorted numerically):

Note that if a user clicks on a numbered tile that is NOT adjacent to the empty space, then nothing should happen.

IMPORTANT: To avoid the having the tiles shuffled into positions that make it literally impossible to move them back into the order shown above, you need to shuffle the tiles carefully. One way to do this is to repeatedly exchange 2 tiles positioned adjacent to the open space in a "legal" way. You should place a **Shuffle** button on the display, as shown in the following solution written by one of the teaching assistants:

You should arrange the initial configuration of the board with the tiles in order, from 1 to 15. Then clicking the **Shuffle** button will reorder the tiles in a solvable manner. You can play with this solution by copying the files `Fifteen.class` and `Fifteen$1.class` from the **unit6** folder on the ***Lecture Examples*** page of our course website.

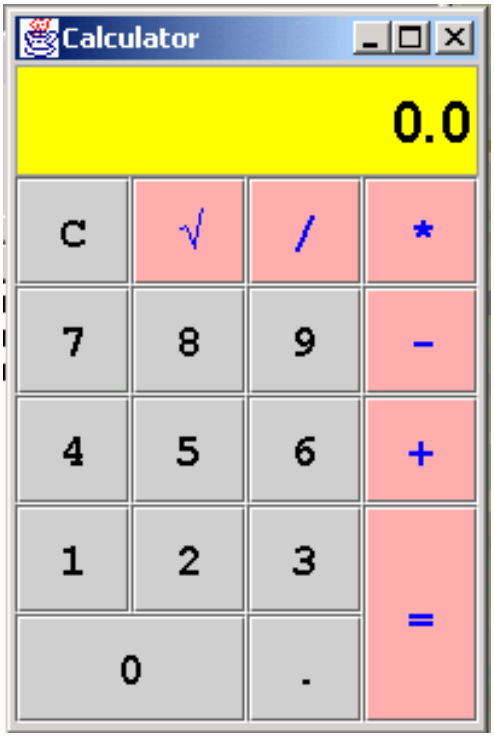
In addition, you should have your program stop and print a congratulatory message when the user has manually clicked the tiles into the correct positions after a **Shuffle** has taken place. (Hint: a straightforward solution will involve a two-dimensional array of **JButton**s.)

[7] 50 points for undergrads, 55 points for grad-credit students

use files Calculator.java, CalcBackend.java, CalBackendTest.java

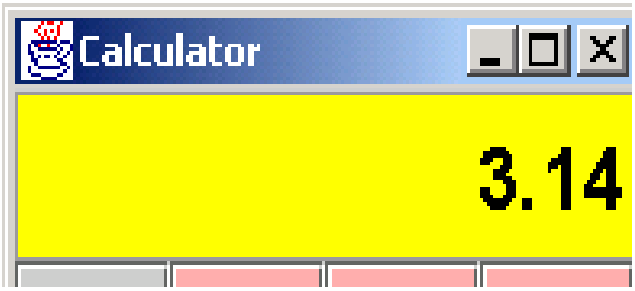
In this problem you will implement a calculator (similar to the one that's built-in to both the Macintosh and Windows OS) using the *Java* programming language. Your program should look something like this when it starts up:

Running on Macintosh OS/X



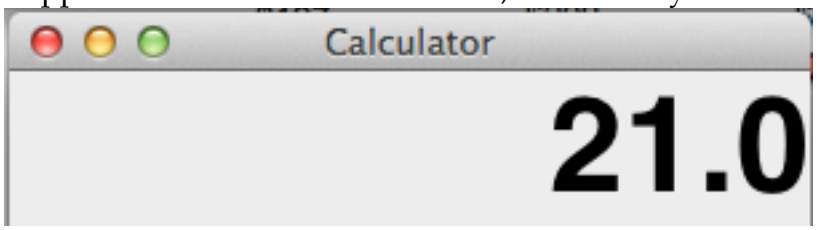
Running on Windows

Although the two calculators look distinctly different because they were run on two different PC platforms, they operate in identical fashion. Note that each of the 18 buttons function by clicking the mouse once on them. Thus, after pressing "3" then "." then "1" then "4", the calculator's display now shows

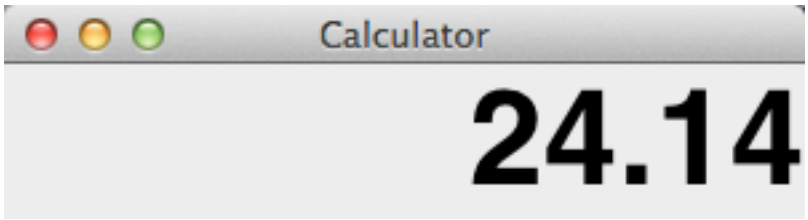


The calculator's main window is implemented as a **JFrame** (a class that is a part of the **javax.swing** package). Each button is an object of type **javax.swing.JButton**.

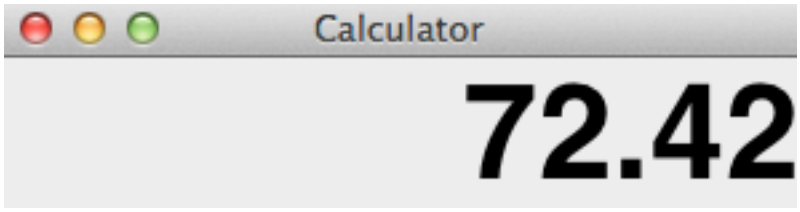
Suppose now I click the "+" button, followed by "**2**" then "**1**"



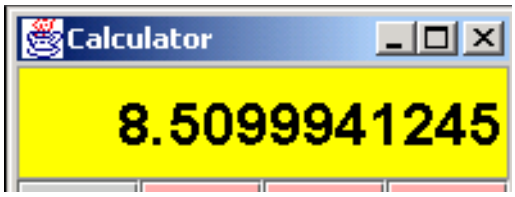
Now I click "=":



And then I click "*" followed by "**3**" and then "="



Now I click "√ "



And I click "√ " once more:

You get the idea: the 4 basic arithmetic operators ("+", "**-**", "*", and "/") are *binary* in nature, requiring two operands. The square-root operator is *unary*, and will perform its computation on whatever value is currently in the calculator's display. If the square-root of a negative number is attempted, the calculator display should produce an ERROR message. In addition, consider what would happen if the user clicks "**1**" then "**6**" then "+" then "**9**" then "√ " then "="… in this case the calculator should take the square root of 9 (which is 3), and then add it to 16, which gives a sum of 19. If the user hit "=" after the 9, and *then* hit "√ " the result 5 would instead be displayed.

Your calculator implementation must make use of the helper class described below which will separate the calculator math logic from the code needed to display it. You must implement a class called **`CalcBackend`**, that keeps track of what should be displayed on the screen and updates this based on key input. The interface for CalcBackend should be:

```
public class CalcBackend {
    //  Constructor
    public CalcBackend() {...}
```

```
    // Button represented by c pressed
    public void feedChar(char c) {...}

    // Returns what the calculator should show
    public double getDisplayVal() {...}
...
}
```

Your **`CalcBackend`** should not have any other public methods or variables. Your main class **`Calculator`** should take care of the *Swing* graphics and have a **`CalcBackend`** instance as a member variable. We have provided you with a unit-test class **`CalcBackendTest`** to check the correctness of your backend implementation. If you have implemented your backend correctly, all the test cases **`CalcBackendTest`** should pass. You should add more test-cases to check the calculator features. You can download the test and a dummy implementation of the backend from the **unit6** folder of the "Java Lecture Examples" page on the course website. The two files you should copy are named ***CalcBackendTest.java*** and ***CalcBackend.java***

NOTE: the above technique of separating display and backend code is widely known as the **Model-View-Controller pattern (MVC)**. In this terminology, **`CalcBackend`** is called the Model, and the *JFrame* code in **`Calculator`** describing the display is called the view, and the ActionListener code in **`Calculator`** that feeds the button pushes to the backend and updates the display is called the controller.

Some miscellaneous cases to be aware of:

[1] If the user clicks an arithmetic operator (e.g., "*") but then immediately clicks a different operator (e.g., "+"), the former operator is discarded and replaced by the more recent one. In other words, addition will get performed in this case. Note that there is no precedence of operators; the evaluation is strictly "left to right."

[2] There is no "unary minus" operator. To create a negative number, you will need to perform a calculation such as subtraction from zero.

[3] The calculator performs all of its computations using the data type **double**. Consider using the *Java* class method

Double.parseDouble(*s*) or **Double.valueOf(*s*).doubleValue()** to convert from a **String** value to its **double** representation.

[4] The display should not allow any part of the number to scroll off either side of the field. And, of course, the user should not be able to input a number with more than one decimal point!

[5] If the "=" button is pressed repeatedly, it should simply re-produce whatever is on the calculator's display.

[6] You can use a **GridLayout** for the **Calculator** buttons, but see extra credit option #7 below.

Extra Credit Options (each is worth a different number of points)

Graduate-credit students must implement ONE of the following for 5 additional points. All students may implement up to 3 options (no more than that) for the listed extra credit point values.

[1] *5 points:* Implement the **MC** (memory clear), **MS** (memory set) and **MR** (memory recall) buttons. Carefully define the semantics of these buttons by playing around with the calculator that is built in to Windows or OS/X.

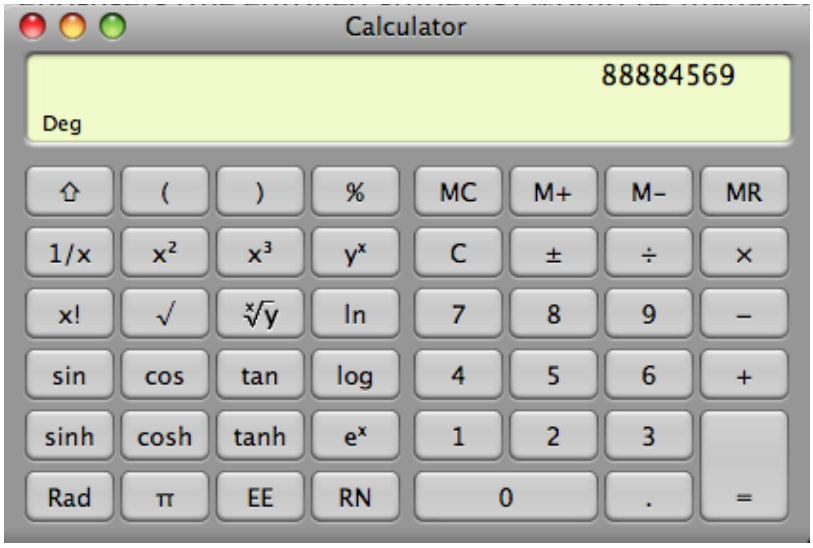
[2] *3 points:* Implement a **+/-** button that *negates* the current display. Clicking this button repeatedly will toggle the value's sign.

[3] *3 points:* Allow the user to control the calculator by using the *keyboard* (in addition to the mouse). In other words, hitting the "*" key will have the same effect as clicking the "*" with the mouse. (Examine the *Microsoft Windows* calculator to find out which keys correspond to √ , **=**, **MC**, and others. One way to do this is to "right-click" on a button and then select "*What's This?*") You'll need to learn how to do this by referring to: https://docs.oracle.com/javase/tutorial/uiswing/events/keylistener.html and https://docs.oracle.com/javase/tutorial/uiswing/misc/keybinding.html

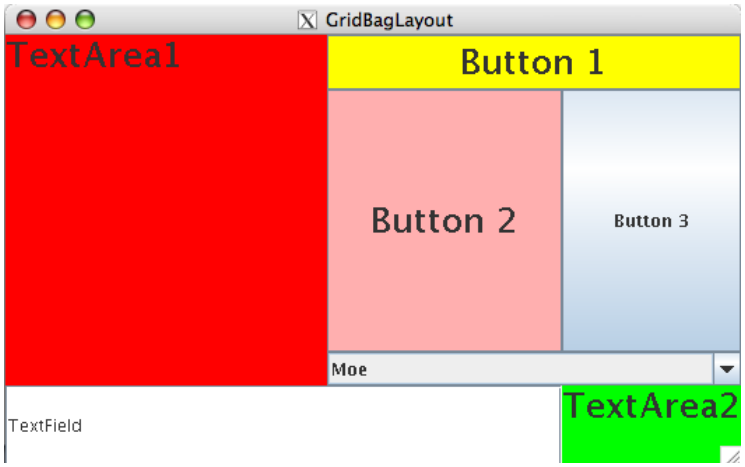
[4] *10 points:* Implement "**(**" and "**)**" buttons that allow the user to parenthesize expressions so that evaluation of arithmetic expressions can be performed in a more controlled fashion.

[5] *5 points:* Open an additional window containing a **JTextArea** with scroll bars so that a "paper tape" of some or all of the user's interaction can be referred to later. There should be a calculator button that controls whether the **JTextArea** is "on" or "off."

[6] *up to 5 points:* Implement **one button** of your own design. Your idea must be approved *in advance* by your teaching fellow. To get you thinking along the right line, here's a view of OS/X's calculator, operating in "scientific mode."



[7] *5 points:* The layout manager used to arrange the buttons in the above figures is the most complicated one (but also the most flexible), a **GridBagLayout**. It allows the calculator window to be resized while keeping the relative size and location of the various buttons intact. Refer to the **GridBagDemo.java** program in an earlier lecture handout for guidance. Study the code in that example to understand how to use the **GridBagLayout** and **GridBagConstraints** classes. That program's output is shown below:

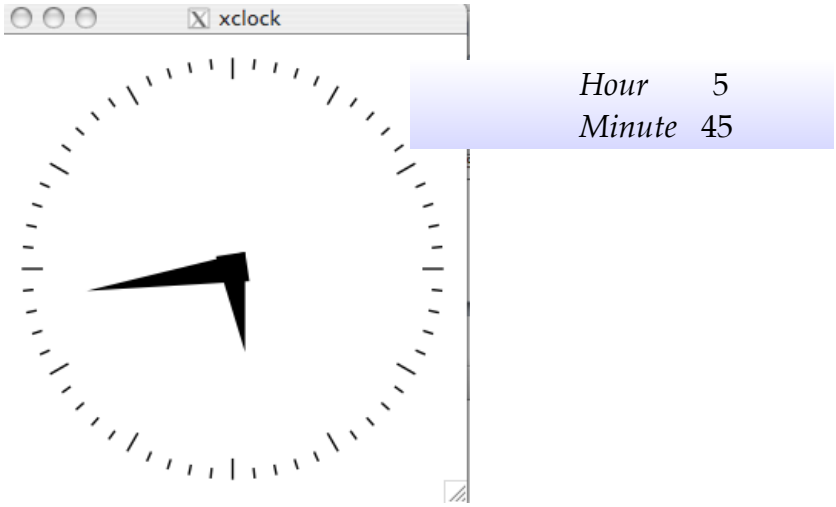


III. Extra Credit Problems (up to 20 points)

[8] 10 points *(for extra credit only)* **use *file named Clock.java***

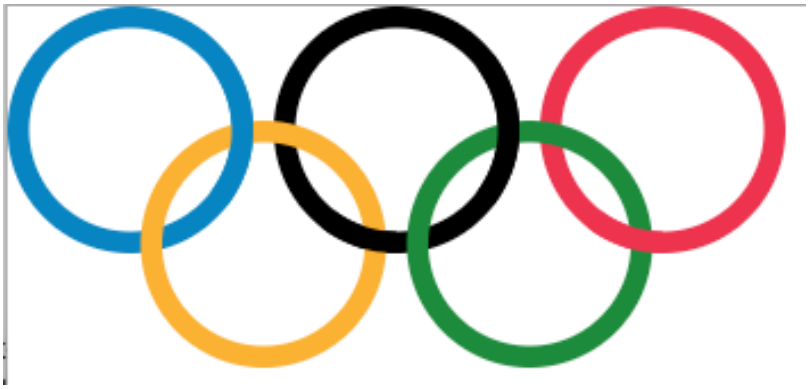
Write a Java application using *Swing* graphics that draws the face of a clock, showing the time that the user enters in two **JTextField** areas (one for the hours, one for the minutes).

Hint: You need to find out the angles of the hour hand and the minute hand. The angle of the minute hand is easy; the minute hand travels 360 degrees in 60 minutes. The angle of the hour hand is harder; it travels 360 degrees in 12 x 60 *minutes*. Don't attempt this problem if you don't remember your trigonometry! Your output could resemble something like this:



[9] 10 points (for extra credit only) **use *file named Olympics.java***

Write a Java application using *Swing* graphics that draws the Olympic rings.



Color the rings in the Olympic colors, as shown here. Your implementation should include a method named **drawRing** that draws a ring at a given position and color (supplied as arguments to the method).

Write a main method that convincingly demonstrates your program in action.

"I only draw with software."