Name: _____

ISTE-121 – Lab 4 – Day 4c
Threads & other things

Objectives

This lab will have you working with threads, and the progress bar.
Remember, you can use any resource you want for completing the labs. What is not allowed is anything that will violate the RIT or IST Academic Dishonesty policy such as copying code. Working in pairs or teams of two or three is encouraged.

NOTE: Based on speeds of the computers, and number of processors, the following lab my work different on different computers. Contact the instructor or TA with questions.

Part 1a – Write a simple Thread

Write a NON-GUI class, Lab2 that has a main, which calls the Lab2 constructor. The constructor creates two instances of the inner class Lab2Inner that extends the thread class. When instantiating the Lab2Inner constructor, pick a name for each thread and pass the name of the thread to the constructor. The inner class's run method prints, "This ran thread" followed by the thread name.
After the Lab2 constructor instantiates and starts both threads, have it print "Program finished". Write the output below.

_____

_____

_____

Part 1b – not so fast, slow down and yield()!

At the beginning of the run() method, place a yield(). What is the output now? (With the multiple CPU's this may not change any results.)

_____

_____

_____

Part 1c – Calculate something

If everything went as expected, the "Program finished" came out first.
With a dual core, it possibly didn't come out first.
To the Lab2 class, add an int attribute that will be a counter, initialize it to zero.
To the "Program finished", print out this out as the "Program finished, count = ", and the counter.
To the end of the run() method in Lab2Inner, add one to the counter.
Now what is the output?

_____

_____

_____

Why was the counter zero?

_____

Part 1d – Wait for the computation to finish, then join the party

To the Lab2 constructor add code between the start and print, that makes that code wait until each of the threads has completed their execution. You may not use the sleep method.
What method did you use? _____
What was the output now?

_____

_____

_____

Have the Instructor / TA check your work at this point.

Instructor / TA: _____

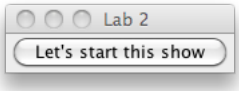
Part 2 – How much progress are you making?

Upon clicking start, two+ threads update the progress bars based on a random number.

This part of the assignment is to help you better understand threads, progress bars, GUI's, and some other strange GUI/Thread interactions. What is shown in Java can be applied to most programming languages, as many react the same way.

Step 1 – Pack your GUI in the Grid

Create a simple GUI, Lab2Part2, whose layout manager is grid layout, of one column and unlimited rows. Add one button with a label you like, this example will use "Let's start this show".

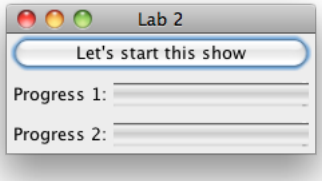


What parameters are used by GridLayout for unlimited rows? _____

If you are wondering what size to make the GUI, stop wondering, use the pack(); statement, rather than setting any sizes. The usefulness of pack() will be evident after we add more components.

Step 2 – Create a classy panel that runs

To the GUI you want to add two or more progress bar lines as shown. But for the overall program to eventually work, we need to add them in a specific way. The label and progress bar components are created within an inner class that extends JPanel and the panel objects are added to the GUI. Since we also need to use threads to move the progress bars, it also needs to be used as a thread. What would it use? Fill in the blanks with what you coded.



class InnerProgress extends JPanel _____ _____

The constructor for this inner class takes a String, such as "Progress #:", and a JProgressBar object. Save these as attributes AND add() these objects to the JPanel. Remember, this class IS a JPanel. Don't write accessors or mutators.

In the main Lab2Part2 constructor, define and instantiate two JProgressBar objects. Have the Lab2Part2 constructor create two objects from InnerProgress, passing in the strings as shown in the example and passing in the JProgresBar objects. Then add these two InnerProgress objects to the GUI. Your GUI should now look like the picture above.

Why can we add InnerProgress objects to the GUI? _____

_____

Step 3 – CLICK THE BUTTON!

Time to make the GUI do something and make sure we have threads that start.

Following where you instantiated objects of the InnerProgress class, use these objects to create Thread objects, and start them. To the run() method, add a print line that says we are running and include the attribute name of the thread.

Run the code, to see if you get the threads to print the lines. If you do not, try to first fix it, then contact the instructor or TA. It should be working.

Create an anonymous inner class attached to the button. That includes the thread object creation and thread starting you wrote earlier in this step. Do not move the InnerProgress object creation, keep that before the anonymous inner class.
Run the code again, CLICK THE BUTTON! It should print the two lines every time you click the button.

Extra learning, if you think you will have time: Move the Thread creation statements to before the anonymous inner class definition. Compile. The error says "local variable … is accessed from within inner class; needs to be declared final". *Defining the thread variable (object) as final, or defining the variable as private attributes to the overall class will fix this. Could try those fixes to practice that they work. Make sure you put the Thread creation statements back into the anonymous inner class before continuing.*

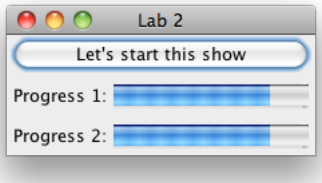
NOTE: The Thread creation must be within the button click event, because a thread cannot be (re)started once it has finished. This way, new threads are created each time you click the button, and these threads are created from the old JPanel/Runnable objects, so they are still referring to the original GUI objects.

I verify the button click properly runs the threads: _____ (you, instructor, or TA)

Step 4 – Don't just sit there with a blank stare on your progress bar, move it!

To the run method, add a 'for' loop that counts `i` from 1 to 80. Within the loop, have it sleep for a random number between 0 and less than 1, multiplied by 100 milliseconds. (Hint: Math class) Then add a `setValue(i);` to set the value of the progress bar. Within the end of the run method, have it again print the thread name, and the finishing millisecond value. For this time you may use `System.currentTimeMillis()`.

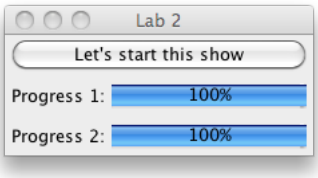
Note that the progress bars only go about 4/5th the way completed. The default values of a progress bar are from 0 to 100. So we have to set the progress bars to go from 0 to 80. In the inner class constructor, set the minimum and maximum values of the JProgressBar to 0 and 80 respectively. Compile and run. It should now appear to go to 100%.



Step 5 – So where are the numbers in the bars?

You can manually place the values there, or use can do it the easy way. In the inner class constructor have the JProgressBar object use,

```
setStringPainted( true );
```



Compile and run. It should show 0% at program startup, and 100% when done. If the numbers only go to 80, why would it show 100%? Discuss this with someone in the class.

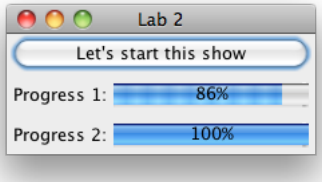
Make sure this step works before continuing.

Step 6 – When the first progress bar stops, everybody stops

So how do we communicate from within a thread, to another thread, when we finish running, and have the other threads stop running? The `.interrupt()` needs to have objects, which one thread doesn't have access, or at least not easy access.

This could be by thread groups, and/or interrupts, but there is a far easier way to do this.

1. Create a boolean attribute called, something like, *keepGoing* in the Lab2Part2 class, set it to *true*.
2. To the 'for' loop in the run method, modify the loop so it will continue looping as long as the number is <=80 and *keepGoing* is true.
3. Immediately after the 'for' loop, set *keepGoing* to false.



Compile and run. Click the button. Observe one progress bar ends at 100%, the other ends very near, but less that 100%.

Click the button a second time. The millisecond messages print. What happens to the progress bars, or the running of the program? Fix this so each time you click on the button the program can properly run without restarting the program.

Step 7 – How much has completed before starting? I'm Indeterminate about that.

Between when the program starts and the button is clicked, we now want the progress bars to show they are indeterminate. This is the Cylon eye movement on a PC, or the lazy barber pole on a Mac.
At the end of the inner class constructor, set indeterminate to true. At the beginning of the run method, set it to false.

Step 8 – Could we have 2 or more threads ending at the same millisecond?

Yes. So we have to determine which one finished first announce it, (System.out.println) and make sure the remaining loser threads are told they didn't finish first. Seems mean, but this is a thread eat thread kind of world.

After the for loop ends, create a synchronized block, or ReenterantLock for lock/unlock. Within this structure, we need to check if we exited by reaching 80, or did the Boolean make us leave the loop? IF the boolean made us leave, that means someone else hit 80

first. For this example we can print out *the thread name +" did not finish first.*" Otherwise, printout *the thread name* + " finished first" and set the keepGoing to false.

Up to here you have covered some of the major concepts of what HW3, requires. However, there is still plenty of work to complete on HW3. Get a sign off here.

Instructor / TA: _____

Step 9 – What!?! There's more? – Yes, if your brain isn't full, this should fill it!

Try this, click on the button several times very fast. All those threads are being started, and running at the same time. Fun, isn't it? Do it some more. Wouldn't it be good to disable the button until after all the threads have completed, and then enable it again? Protecting the program from users who click the button several times very fast.

At the end of the button action performed method, set the button disabled:

```
___.setEnabled( false ); // disables the button
_____.join();            // wait for thread 1 to end
_____.join();            // wait for thread 2 to end
___.setEnabled( true );  // enables the button
```

Compile & Run it. What happens?

_____

This happens because the GUI does not get updated while the actionPerformed method is still running. The method needs to exit for the GUI to update. We need to have the join and enabling to happen independent of the actionPerformed method. We will do this by creating another thread for the joins and setEnabled. Look carefully at the following code, and as you enter it, **comment what it does** in the code. Any variables with __ around them must be **substituted with your object names**.

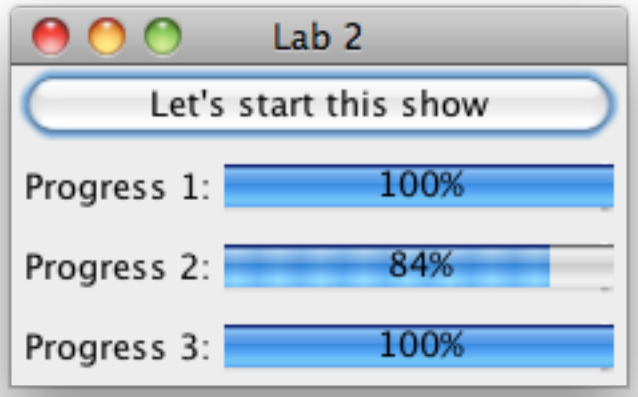
```
__button__.setEnabled( false );
Thread extra = new Thread(){
     public void run() {
          try {     __thread1__.join();
                    __thread2__.join();
          } catch( InterruptedException ie ){}
          __button__.setEnabled( true );
     }
};
extra.start();
```

Step 10 – Add 1, 2 or more progress bars, see what happens.

Let's add more progress bars to the GUI, but not quite yet.
Before adding the code, write here:

- All the objects needed to be created

- What places need to have code added



Now add the code and get it working. How close were you to your original thoughts? Notice the items you missed, so you can remember them next time.

Last Question:
In the above picture, two progress bars ended at the same time. What is the expected output regarding which thread "finished first", and which "didn't finish first"?

The very impressed Instructor or TA initials: _____