CST8221 – Java Application Programming
Hybrid Activity #11
Swing GUI and Threads

Terminology

As you already know one of the reasons to use threads in your applications is to make your programs more responsive delegating the time consuming task to worker threads. However, you must be very careful what you do in a worker thread. As you are aware Swing is not thread-safe. Not thread-safe means that if you try to manipulate user interface components from multiple threads, your user GUI can become corrupted.

The Nature of Things

When you use mix Swing GUI with threads you should follow tow simple rules:

RULE# 1

*If an action takes a long time to complete, do it in a separate worker thread, and never in the **event dispatch** thread.*

RULE# 2

*Do not manipulate Swing components in any thread other than the **event dispatch thread**.*

The reason for the first rule is evident. If you take a long time in the event dispatch thread, the application will not be responsive because it cannot react to any current event. In particular, the event dispatch thread should never perform input/output operation, which might block indefinitely, and it should never call the thread sleep() method.
The second rule is known as the “single thread rule” for Swing programming.

If you read the rules carefully it appears that they are in conflict with each other. Suppose that you spawn a separate thread to run a time-consuming task. As a result of the operation, you usually want to update the some component in the user interface to indicate progress while your thread is working. When your task is finished, you want to update the GUI again. But you cannot manipulate Swing components from your thread.
To solve this problem and satisfy both rules, you can use two utility methods in any thread and add arbitrary actions to the event dispatch thread. For example, you want to periodically update a label component in a thread to indicate some progress. According to Rule 2, you cannot call *label.setText()* method from your thread. Fortunately, *use invokeLater()* and *invokeAndWait()* methods of the *EventQueue* class to have that call executed in the event dispatching thread (remember, I have used the first method many times in the example code I have given to you).

Here is an example of what you have to do to solve the problem. You have to place the code affecting the Swing components into the *run()* method of a class that implements the *Runnable* interface. Then you have to create an object of that class and pass it to the static *invokeLater()* or *invokeAndWait()* method. For example, here is how to update a label text:

```
public class MyWorkerClass implements Runnable{
//...
 public void run () {
   // some code doing the work of the worker thread
   //...
   // it is time to update the label
   EventQueue.invokeLater(new Runnable() {
        public void run(){
         label.setText("progress update value");
        }
   });
 // some more code
 }//end thread run method
}//end class
```

It is important to understand that the *invokeLater()* method returns immediately when the event is posted to the event queue. The *run()* method is executed asynchronously. The *invokeAndWait()* method waits until the run method has actually been executed (see the *EventQueue* class documentation). It is also important to understand that both methods execute the run method in the event dispatch thread. No new thread is created.
In the situation of updating the progress label, *the invokeLater()* method is more appropriate. Users would rather have the worker thread make more progress than have the most precise progress indicator.
This is the Java “classic way” to solve the problem. There is another way - using the convenience **SwingWorker** class from the ***javax.swing*** package (see Lab 12).

Note: JavaFx is not tread-safe either. The single thread rule must be applied to JavaFX multi-threaded GUI applications as well. You can find the *runLater()* method in the Platform class.

References

Textbook 1 – Chapter 23.
“Core Java – Volume I – Fundamentals” by G.S Horstman and G. Cornell, Prentice Hall

Code Example

You will find the code examples in ***CST8221_HA11_code_examples.zip***.
The code example demonstrates how to use the *invokeLater()* method to safely modify the contents of a combo box component. Remember that JComboBox turned generic in Java 1.7?

Exercise

Run the **SwingThreadTest** application.
First click the Bad button. Click the combo box a few times. Move the scroll bar of the combo box. Click the Bad button again. Keep clicking the combo box. At some point you will see a lot of exceptions generated in the console window. One of them is *ArrayOutOfBounds* exception. The reason for this exception is the corruption of the combo box data by the running thread – remember Swing is not thread-safe.
Stop the application and run it again. Now click the Good button. Click the combo box a few times. Move the scroll bar of the combo box. Click the Good button again. Keep clicking the combo box.

Do not click the Bad button. Noting bad happens. Problem solved!
Once you see how they work, explore very carefully the code, and try to understand its inner workings.

Questions

Q1. Is Swing thread-safe?
Q2. Does *the invokeLater()* method create a new thread?
Q3. Does *the invokeLater()* method return immediately or does it wait for *run()* to complete?
Q4. Is JavaFX thread-safe?

Submission

No submission is required for this activity.

Marks

No marks are allocated for this activity, but remember that understanding how *the demo example works* is very important if you want to mix Swing with threads.

And do not forget to ***thread*** *safely the dark alleys of Swing and FX…*