Thread 3 GUI

Handout written by Nick Parlante

GUIs and Threading

Problem: Swing vs. Threads

- How to integrate the Swing/GUI/drawing system with threads?
- Problem: The GUI system -- components, buttons, frames, etc. is a big data structure which must support many read and write operations. Cannot have two or more thread manipulating that data structure at the same time (unless they are all just reading it).
- e.g. read/write conflict if paintComponent() reads the GUI state while another thread changes it

Solution: Swing Thread / Event Queue aka One Big Lock

- There is one, designated official "Swing thread"
  - Also knows as the "draw thread" or the "UI thread" or the "event thread"
- The system does all Swing/GUI notifications using the Swing thread, one at a time..
  - paintComponent() -- always called on the Swing thread
  - All notifications -- action events, mouse motion events -- are called on the Swing thread. e.g. actionPerformed() is called on the Swing thread.
- The system keeps a queue of "Event jobs". When the swing thread is done with its current job, it loops around and gets the next one and does it. The Swing thread does one thing at a time. It is approximately FIFO, but not exactly. For example, the Swing thread might process button clicks ahead of draw requests.
- Only the swing thread is allowed to access (read or write) the state of the GUI -- the labels of things, the geometry, the nesting, the listeners, etc. of the swing components. e.g. getText(), setLayout(), ... all those can only be called by the swing thread. This is a big constraint -- only the Swing thread may touch swing objects.
- Since only the Swing thread is allowed to access swing state, in effect there is one big lock over all the swing state.

Common GUI Thread Solution

- This solution -- identifying a special thread which exclusively "owns" all the GUI state -- is the most common way to integrate a GUI with threading.
- It is probably the simplest, best way to combine a GUI with threading.

Programmer Rules...

1. On the swing thread -- using Swing is ok

- When you are on the Swing thread, you are allowed to using Swing all you want.
- All paint and control notifications sent to your code -- paintComponent(), actionPerformed(), ... -- these are sent on the Swing thread, so you are fine in those cases.
- e.g. fine to call container.add(), setPreferredSize(), setLayout(), getText(), setText()
- This is how all our solutions have worked thus far without knowing anything about threading -- we were always running on the Swing thread.

2. Don't hog the swing thread

- Do not do time-consuming operations on the swing thread
- There is only one swing thread, and while you are running on it, it cannot do any of the normal housekeeping work of the GUI. No Swing/GUI event processing, drawing, etc. will happen until you finish your processing and return the swing thread to the system -- then it can de-queue drawing requests, mouse events, push in buttons, etc.
  - Note in the example below how "Homework Cancelled" never appears on screen.
- "Returning the swing thread to the system"
  - e.g. your code is actionPerformed() { foo(); bar(); }, running on the Swing thread
  - After foo() and bar() are done and the method exits its last } ... this returns the Swing thread to dequeue the next task.
- Fork off a worker thread to do a time-consuming operation, leaving the Swing thread to maintain the GUI.

3. Worker not on the swing thread -- no touch Swing state!

- A thread which is not the swing thread may not send messages that use the swing state (add(), getText(), setText(), setBounds(), ...).
- Instead, use invokeLater() (below) to run code on the Swing thread
- Violating this rule will appear to work. However, the violation is a latent bug/race-condition between your thread and the Swing thread, waiting to screw up the data someday.
- Exceptions: a few rare swing methods use locks internally so that they can be called by any thread -- they are "thread safe". Thread safe methods include repaint(), revalidate(), and the add/remove Listener methods. Finally, as a 1-off exception, setText() of the JTextField is thread-safe. Overall, the vast majority of Swing methods are not thread safe.
- Another exception is before the component has been brought on screen (setVisible(true)) -- before the component is on screen, the swing thread is not using it, so it's ok to add(), setBounds(), etc. on it.
  - This is often used in main() ... set up Swing state (not on the Swing thread), and call setVisible(true) as the last step
  - The official best practice for Java is being revised so the main()/setVisible(trick) is no longer recommended, although it should continue to work.
  - Instead, main could look like: SwingUtilities.invokeLater(new Runnable() { public void run() { <set up swing state here> } });
  - This new requirement is not a client friendly design -- make common things easy, remember, and setting up the GUI is a very common, ordinary thing to want to do. I hope they come up with a better way for no-brainer client code to bring up a GUI simply.

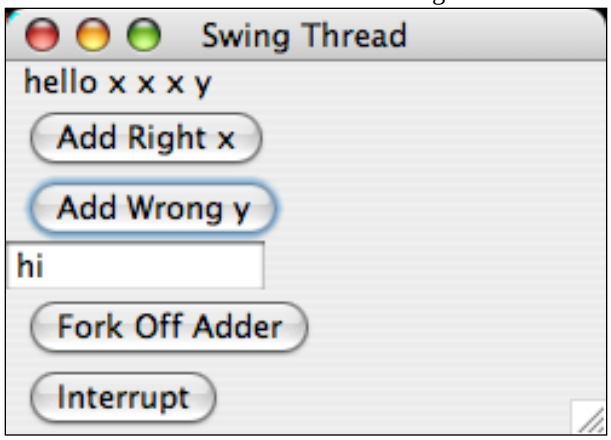
SwingUtilities

- Built in utility methods that allow you to "post" some code to the swing thread to run later.
- "Runnable" interface -- defines public void run() { }
- SwingUtilities.invokeLater(Runnable)
  - Queue up the given Runnable -- the Swing thread will execute the Runnable when the Swing thread gets to it in the queue of things to do.
- SwingUtilities.invokeAndWait(Runnable)
  - As above, but block the current thread (a worker thread of some sort presumably), until the Runnable exits.
- In Java 5, this class is now called "EventQueue" instead of "SwingUtilities" but the functionality is the same.
- Java 6 adds a SwingWorker class which is another way for a worker thread to communicate with the GUI. It's more powerful, but it's a little more complicated. For my examples, I'll use the simple

invokeLater() technique. In any case, the key point is to understand that a worker thread cannot manipulate GUI directly.

SwingThread Demo

- Demonstrates swing thread issues
- Hogging the swing thread -- bad!
- Fork off a worker + worker uses SwingUtilities.invokeLater() to communicate back to the swing state




```
// SwingThread.java
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

/*
 Demonstrates using workers and the swing thread,
 and interruption.
*/
class SwingThread extends JFrame {
    private JLabel label;
    private JButton right;
    private JButton wrong;
    private JButton fork;
    private JTextField field;
    private LabelWorker worker;
    private JButton inter;

    public SwingThread() {
        super("Swing Thread");

        setLayout( new BoxLayout(getContentPane(), BoxLayout.Y_AXIS) );

        // put in a label
        label = new JLabel("hello");
        add(label);

        // Button that adds an "x" when clicked -- fine,
        // runs on the swing thread
        right = new JButton("Add Right x");
        add(right);
        right.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                label.setText(label.getText() + " x");
            }
        });

        // Bad -- hogs the Swing thread
        // the whole GUI will appear to lock up for 5 seconds -- use worker thread instead
        // Note also that the "All Homework Is Cancelled" NEVER appears on screen
        wrong = new JButton("Add Wrong y");
```

```
    add(wrong);
    wrong.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            String text = label.getText();
            label.setText("All Homework Is Cancelled!");
            // sleep for 5 seconds -- simulate time consuming operation
            try {
                Thread.sleep(5000);
            } catch (InterruptedException ignored) {}
            label.setText(text + " y");
        }
    });

    // Field appends text to label -- uses worker thread correctly
    field = new JTextField("hi", 20);
    field.setMaximumSize(new Dimension(200, 20));
    add(field);

    worker = null;
    fork = new JButton("Fork Off Adder");
    add(fork);

    // fork button -> set text, then fork off worker
    fork.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            // set text right away -- ok, we're on the swing thread
            label.setText(label.getText() + " ... WAIT FOR IT!");

            // Interrupt previous worker if it exists
            if (worker != null) worker.interrupt();

            // fork off new worker using text from field
            worker = new LabelWorker(field.getText());
            worker.start();
        }
    });

    // Interrupt existing worker
    inter = new JButton("Interrupt");
    add(inter);
    inter.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            // Q: Is there a race condition between this code and
            // the above worker interrupt code?
            if (worker != null) {
                worker.interrupt();
                worker = null;
            }
        }
    });

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    pack();
    setVisible(true);
}
```

```
    // Takes a word, appends it 10 times to the label,
    // working very slowly. Exits when interrupted
    private class LabelWorker extends Thread {
        String word;

        public LabelWorker(String initWord) {
            word = initWord;
        }

        public void run() {
            String text = "";
            for (int i = 0; i < 10; i++) {
                // Sleep for a second, exit on interrupt
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException ex) {
                    // Control goes here if we are interrupted while sleeping --
                    // exit the run loop.  (Way 1 to notice interruption)
                    break;
                }

                // Do some computation, storing in a final temp variable,
                // so it is visible in the Runnable
                text = text + word;
                final String finalText = text;

                // Notice if interrupted -- exit
                // (Way 2 to notice interruption)
                if (isInterrupted())
                    break;

                // NO NO NO, cannot do this
                // label.setText(finalText);

                // Message back to the GUI using invokeLater/Runnable
                SwingUtilities.invokeLater(new Runnable() {
                    public void run() {
                        label.setText(finalText);
                    }
                });
            }
        }
    }

    public static void main(String[] args) {
        new SwingThread();
    }
}
```