Graphical User Interfaces (2)

(and Event Driven Programming)

CS 2110

Announcements

- **Quiz 6** *[does not include GUI content]*
  - **Open**: @3:30pm on 03/04/2020 (Wednesday)
  - **Close**: @11:30pm on 03/06/2020 (Friday – **Today!**)
- **Homework 3**: due tonight (Friday) by 11:30pm – *submit on Web-CAT!*
- **Homework 4**: coming soon… based on **GUIs** (event-driven programming)!
- Friendly reminder:
  - When sending email please include "CS 2110" in your email subject header
  - We all teach other classes, and receive a lot of email… doing this will help ensure we can identify and respond to your email
  - If we don't respond right away, don't hesitate to send a follow-up email!

Swing Framework

Swing: in the `javax.swing` and `java.awt` packages

- GUI elements in classes that begin with "J"
  - `JFrame`, `JPanel`, `JLabel`, `JButton`, `JTextArea`, etc
  - `https://web.mit.edu/6.005/www/sp14/psets/ps4/java-6-tutorial/components.html`
- Layout classes
  - `BorderLayout`, `FlowLayout`, `GridLayout`, etc
  - `https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html`
- Event handlers
  - "Listeners" for mouse, button, and key presses, etc

Reminder: Event Driven Programming

- Event-driven programming is a **programming paradigm** in which the flow of the program is determined by **events** such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs/threads
- Event-driven programming is centered on performing certain actions in response to **user input**
- In an event-driven application, there is generally a **main loop** that **listens for events**, and then triggers a **callback function** when one of those events is detected (more on this a bit later)

Inner Classes - Encapsulation

- It is legal in Java to define **a class within a class**
- Inner classes offer a form of encapsulation that make sense if they are only used in one place
  - Part of the outer class' implementation

```
class OuterClass {
   // has access to OuterClass objects' fields and methods
   private class InnerClass {...}
-or-
   // can exist outside of OuterClass objects
   static class StaticInnerClass {...}
}
```

Example of an Inner Class - ButtonListener

- Create an **inner class** that **implements** the **ActionListener interface**

```
// create an inner class for the button action
class ButtonListener implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      if (e.getActionCommand().equals("click")) {
         // ButtonListener triggered and the command was “click”
         infoLabel.setText("Button clicked"); // DO STUFF!
      }
   }
}
```

ActionListener (the *Interface*) requires <u>one</u> method: **actionPerformed**, which accepts an **event** object as a parameter. This class holds the code that gets executed when the button is clicked (here, a *label* is updated).

Inner Class and associated Button code

- In "`addComponentsToPane()`" method in "`GUIDemo.java`" file

Inner Class

```
// create an INNER class for the button action ------------------
class ButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("click")) { // action command matches ("click")
            infoLabel.setText("Button clicked");
        }
    }
} // END inner class ButtonListener ------------------------------


// button
actionButton = new JButton("Action");
actionButton.setActionCommand("click");
actionButton.addActionListener(new ButtonListener());
panel1.add(actionButton);
```

Anonymous Classes – can be created inline!

- Defined following the `new` keyword
- Must be based on an existing type/interface
  - E.g. `someMethod( new ActionListener() { . . . } );`
- **It has no name**, so you *cannot* use it *elsewhere*
- Useful when you need a class ***once***, to perform a *specific* duty

See `GUIDemo_Anon.java` for an example!
(next slide gives an example of this
replacing the ButtonListener inner class)

```
//        // create an INNER class for the button action ----------------------
//        class ButtonListener implements ActionListener {
//            public void actionPerformed(ActionEvent e) {
//                if (e.getActionCommand().equals("click")) { // action command matches ("click")
//                    infoLabel.setText("Button clicked");
//                }
//            }
//        } // END inner class ButtonListener ---------------------------------


        // ### Example of using an Anonymous Class ###
        // (Note, we comment out the ButtonListener inner class that we had previously.)

        // button
        actionButton = new JButton("Action");
        actionButton.setActionCommand("click");
        actionButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if (e.getActionCommand().equals("click")) { // action command matches ("click")
                    infoLabel.setText("Button clicked");
                }
            }
        } );
        panel1.add(actionButton);
```

Anonymous Class

Concurrency (a preview)

- GUI environments support *concurrent processes* (threads)
- Java spawns a new **thread** that allows your program to do two (or more) things at once
- Threads execute **Runnable** tasks
- Create a **runnable** task by implementing the **Runnable** interface:

```
public interface Runnable {
      public void run();    // has only one method stub
}
```

Concurrency (a preview)

- You'll see this in the code, but do not worry about the details right now.

```
public static void main(String[] args) {
    // Example of an anonymous class - "injected" directly in the location it is needed
    javax.swing.SwingUtilities.invokeLater(new Runnable() { // used in conjunction with threads
        public void run() {     // a thread created by a GUI element looks for the 'run' method
            createAndShowGUI();
        }
    });

}
```

From GUIDemo.java : threads require run()

(note: this is also written using an anonymous class)

Concurrency (a preview)

- Tasks (like our GUI) can be scheduled using the Event Dispatcher's **invokeLater** method
  - **invokeLater** is a static method that accepts a **Runnable** object

- Many threads are running on a system. When the GUI process is scheduled, **invokeLater** allows the GUI application to execute specific commands (the commands found in the run() method) at an appropriate time.

- *Note: we will NOT focus on concurrency right now (an up-coming topic!)*

Eclipse Examples

- `GUIDemo.java // GUIDemo_Anon.java`
  ~ showing anonymous class example
- `Button_SQRT_Listener.java` *~ adding listener to the button*
- `ColorWindow.java` *~ multiple buttons, different actions*
- `Parrot.java` *~ parrots back what you say!*

Good Resources

- Your **book** (Chapter 20)
- **MIT Guide** (*dated, but good*): https://web.mit.edu/6.005/www/sp14/psets/ps4/java-6-tutorial/components.html
- Information on **JFrames** http://www.tutorialspoint.com/swing/swing_jframe.htm
- Information on **ActionListeners** http://www.tutorialspoint.com/swing/swing_action_listener.htm
- As always, you can find **code examples** (relating to Java Swing and GUIs) under Collab Resources

In-Class Activity: GUI Day 2

- Previously we asked you to create a window that contained:
  - A text field, two labels, and a button

- Now, add support for the **ActionListener** interface
- Add an **action** that, when the **BUTTON** on your form is clicked, it converts an amount entered into the **TEXT FIELD** from kilometers to miles and updates the **result LABEL**
- Also update your **instruction LABEL** so it is appropraite for this situation
- *Hint:* there are 1.609344 kilometers to 1 mile