Programing Languages
and Techniques
(CIS120e)

Lecture 29
Nov. 19, 2010

Swing I

Why study GUIs (again)

• Most common example of event-based programming
• Heavy and effective use of OO inheritance
  – Nice opportunity to compare our “hand-rolled objects” in OCaml with those supported by Java’s rich object system
• Case study of library organization

Event-driven programming

• **Passive**: Application waits for an event to happen in the environment
  – When an event occurs, the application responds to the event, and then waits for the next event.
  – The program’s flow of control is dictated by the stream of events arriving from the outside

• Examples:
  – *A graphical, window-based user interface*
  – Web server
  – Control system (thermostat, robot)

Swing

• Java library for GUI development
  – `javax.swing.*`
• Built on existing library: AWT
  – `java.awt.*`
  – If there are two versions of something, use Swing’s. (e.g., `java.awt.Button` vs. `javax.swing.JButton`)
  – The “Jxxx” version is usually the one you want, rather than “xxx”.
• Portable
  – Communicates with OS’s native window system
  – Same Java program looks slightly different when run on your PC and my Mac.
Background Reading

• Java Tutorial on “Creating a GUI With JFC/Swing”
  – For starters:
    • Getting Started with Swing
    • Using Swing Components

Swing: Two Types of Components

• Atomic components
  – Interact directly with user
  – Examples:
    • Buttons
    • Checkboxes
    • TextFields
    • RadioButtons
    • …

Swing: Two Types of Components

• Atomic components
• Containers

Fundamental class: JComponent

• Stores attributes common to GUI elements
  – size (height/width)
    • preferred size
    • minimum/maximum size
  – location
  – font
  – foreground/background color
  – border
  – what is visible
  – many more…
Swing: Two Types of Components

• Containers
  Hold and position other components
  
  1. *Top-level* containers
     – Lays out screen real estate
     – JFrame, JDialog
  
  2. *Intermediate* containers
     – Organize and position other components
     – JPanel

JFrame

• Components in a frame are stored in its “content pane”

```java
import java.awt.*;
import javax.swing.*;

class FrameDemo {
  JFrame f = new JFrame(“FrameDemo”);
  JButton b = new JButton(“Press”);
  Container cp = f.getContentPane();
  cp.add(b);
  f.setVisible(true);
}
```

JFrame

• Represents a top-level window
  
  • *Displayed directly by OS* (Mac, PC, etc.)...
    – Title
    – Border
    – *Therefore, Not* a subclass of JComponent
  
  • Displayed by Java within window...
    – menu bar (optional)
    – user-specified components (in content pane)
  
  • Can be moved, resized, iconified, closed.

Panes

• Other Panes exist beyond ContentPane
  
  • We don’t care right now....
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class OnOffSwitch extends JFrame {
    public OnOffSwitch () {
        super("On/Off Switch");  // frame title

        // create button and set its colors
        JButton button = new JButton("On/Off");
        button.setForeground(Color.black);
        button.setBackground(Color.red);

        // add button to content pane
        this.getContentPane().add(button);
        this.setSize(300,300);
        this.setVisible(true);
    }
}

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

Making the Button do something

- So far, pressing the button has no effect at all.
- We’ve created a button, but haven’t “wired it up” to do anything.
  - When the button is pressed, it generates an ActionEvent.
  - We can program the button with an ActionListener to respond to the ActionEvent.
What’s an ActionEvent?

- ActionEvents are objects.
  - subclass of abstract class java.awt.AWTEvent.
- If user clicks on a button, it generates an ActionEvent.
- More generally, ActionEvents are generated by the user doing things to the GUI (mouse clicks, keypresses, ...)

Switcher

class Switcher implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        Component source = (Component)e.getSource();
        Color oldForeground = source.getForeground();
        Color oldBackground = source.getBackground();
        source.setForeground(oldBackground);
        source.setBackground(oldForeground);
    }
}

- Register listener with the button:
  ```java
  button.addActionListener(new Switcher());
  ```

ActionListener

- An ActionEvent triggers an ActionListener.
  - Causes the actionPerformed method of the ActionListener to be called.
- Our job is to program the actionPerformed method.
  ```java
  interface ActionListener {
      void actionPerformed(ActionEvent e);
  }
  ```
- To activate, just register listener with the button.
  ```java
  button.addActionListener(listener);
  ```

Results

Before Clicking Button

After
Putting the button directly in the JFrame makes it fill the whole frame—not very attractive

To let it take its natural size, we create a JPanel that contains the button

More Buttons...

- A JPanel can contain many subcomponents
- The layout of these subcomponents can be specified in very flexible ways
- By default, subcomponents are filled in left-to-right, with multiple rows as necessary

```java
JPanel panel = new JPanel();
panel.add(button);
this.getContentPane().add(panel);
```

Extending the example

```java
// create another button and set its colors
JButton button2 = new JButton("On/Off");
button2.setForeground(Color.black);
button2.setBackground(Color.green);
// And associate an action
button2.addActionListener(new Switcher());

// Add both buttons to a panel
JPanel panel = new JPanel();
panel.add(button);
panel.add(button2);
this.getContentPane().add(panel);
```
More interesting actions

- The ActionListener attached to a button can do things to other components of the GUI
- For example, let’s create a label next to the button that displays its current state (on/off)...
- Useful facts:
  - Events can trigger multiple ActionListeners
  - Events of one component can easily affect other components through Listeners

```java
class Switcher2 implements ActionListener {
    JLabel mytext;

    Switcher2 (JLabel l) { mytext = l; }

    public void actionPerformed (ActionEvent e) {
        Component source = (Component)e.getSource();
        Color oldForeground = source.getForeground();
        Color oldBackground = source.getBackground();
        source.setForeground(oldBackground);
        source.setBackground(oldForeground);

        String current = mytext.getText();
        if (current.equals("off")) mytext.setText("on");
        else mytext.setText("off");
    }
}
```

```java
// create button
JButton button = new JButton("On/Off");
button.setForeground(Color.black);
button.setBackground(Color.red);
// create associated text field
JLabel text = new JLabel("off");
// And associate an action
button.addActionListener(new Switcher2(text));

// add to content pane inside a panel
JPanel panel = new JPanel();
panel.setBackground(Color.yellow);
panel.add(button);
panel.add(text);
this.getContentPane().add(panel);
```

Results

Before

After
More flexible: WindowListener

- Closing windows creates a WindowEvent
  - React to that event by terminating program

```java
public interface WindowListener extends EventListener {
    void windowActivated(WindowEvent e);
    void windowClosing(WindowEvent e);
    void windowClosed(WindowEvent e);
    void windowDeactivated(WindowEvent e);
    void windowDeiconified(WindowEvent e);
    void windowIconified(WindowEvent e);
    void windowOpened(WindowEvent e);
}
```

Problem

- Problem: When window is closed, program does not terminate.
- How to fix it?

- Easy way:
  ```java
  frame.setDefaultCloseOperation(EXIT_ON_CLOSE);
  ```

Terminator class

```java
class Terminator implements WindowListener {
    public void windowClosing(WindowEvent e) {
        Window w = e.getWindow();
        // delete top-level frame, release OS resources,
        // generate WindowClosed event
        w.dispose();
    }

    public void windowClosed(WindowEvent e) {
        // terminate the application
        System.exit(0);
    }

    // other five methods do nothing
    void windowActivated(WindowEvent e) {}
    void windowDeactivated(WindowEvent e) {}
    void windowDeiconified(WindowEvent e) {}
    void windowIconified(WindowEvent e) {}
    void windowOpened(WindowEvent e) {}
}
```
**Register WindowListener**

```java
public OnOffSwitch () {
    super("On/Off Switch"); // frame title
    this.addWindowListener(new Terminator());
    // create button and set its colors
    JButton button = new JButton("On/Off");
    button.setForeground(Color.black);
    button.setBackground(Color.white);
    // create and register button’s listener:
    button.addActionListener(new Switcher());
    // add button to JFrame’s content pane:
    this.getContentPane().add(button);
}
```

**Problem**

- A window listener must implement all 7 methods of `WindowListener` interface
- We only care about 2 of those methods → lots of annoying boilerplate
- Solution: `WindowAdapter`

**Adapter classes:**

- Swing provides a collection of abstract event adapter classes
- These adapter classes implement listener interfaces with empty, do-nothing methods
- To implement a listener class, we extend an adapter class and override just the methods we need

**Terminator class (II)**

```java
class Terminator extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        Window w = e.getWindow();
        w.dispose();
    }

    public void windowClosed(WindowEvent e) {
        System.exit(0);
    }
}
```