CSE 331
SOFTWARE DESIGN & IMPLEMENTATION
GUI & (A LITTLE ON) DESIGN PATTERNS III

Autumn 2011

Why learn GUIs?

- Learn about event-driven programming techniques – perhaps the most-used version of inversion-of-control
- Practice learning and using a large, complex API
- A chance to see how it is designed and learn from it (design pattern usage, etc.)
- Caution: There is a ton of information for GUI programming – huge APIs
  - You won't memorize it all; you will look things up as you need them
  - But you have to learn the fundamental concepts and general ideas

Don't mistake…

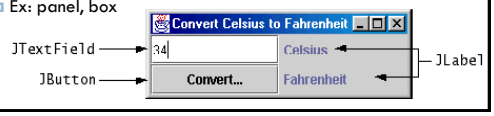
- … how to build a GUI well with …
- … what is a good UI for people
- Just another version of "building the system right vs. building the right system"
- We'll come back to some usability issues – much more related to "building the right system" later in the term

Java GUI History

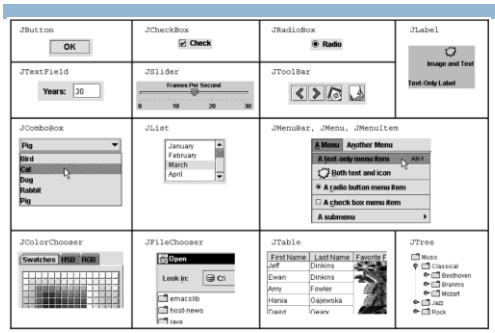
- Abstract Windowing Toolkit (AWT): Sun's initial effort to create a set of cross-platform GUI classes (JDK 1.0 - 1.1)
  - Maps general Java code to each operating system's real GUI system
  - Limited to lowest common denominator; clunky to use
- Swing: A newer GUI library written from the ground up that allows much more powerful graphics and GUI construction (JDK 1.2+)
  - Paints GUI controls itself pixel-by-pixel rather than handing off to OS
  - Better features, better compatibility, better design
- Warning: Both exist in Java now; easy to get them mixed up; still have to use both in various places

GUI terminology

- *window*: A first-class citizen of the graphical desktop
  - Also called a top-level container
  - Ex: frame, dialog box, applet
- *component*: A GUI widget that resides in a window
  - Also called controls in many other languages
  - Ex: button, text box, label
- *container*: A logical grouping for storing components
  - Ex: panel, box



Swing components (partial)



Swing inheritance hierarchy

- Component (AWT)
  - Window
    - Frame
      - JFrame (Swing)
      - JDialog
  - Container
    - JComponent (Swing)
      - JButton
      - JComboBox
      - JMenuBar
      - JPopupMenu
      - JScrollPane
      - JSplitPane
      - JToolbar
      - JTextField
      - JColorChooser
      - JLabel
      - JOptionPane
      - JProgressBar
      - JSlider
      - JTabbedPane
      - JTree
      - ...
      - JFileChooser
      - JList
      - JPanel
      - JScrollbar
      - JSpinner
      - JTable
      - JTextArea

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

Component properties

- Each has a `get/is` accessor and a `set` modifier
- Ex: `getColor`, `setFont`, `setEnabled`, `isVisible`

name	type	description
background	`Color`	background color behind component
border	`Border`	border line around component
enabled	`boolean`	whether it can be interacted with
focusable	`boolean`	whether key text can be typed on it
font	`Font`	font used for text in component
foreground	`Color`	foreground color of component
height, width	`int`	component's current size in pixels
visible	`boolean`	whether component can be seen
tooltip text	`String`	text shown when hovering mouse
size, minimum / maximum / preferred size	`Dimension`	various sizes, size limits, or desired sizes that the component may take

`JFrame`

A window holding components

- Constructors with an optional title
  `public JFrame()`
  `public JFrame(String title)`
- Make a frame `f` appear on the screen
  `f.setVisible(true) public void`
- Place the given component or container inside the frame `f`
  `f.add(Component comp)`
- Make the frame perform a given action when it closes
  `public void setDefaultCloseOperation(int op)`
  - Common value passed: `JFrame.EXIT_ON_CLOSE`
  - If not set, the program will never exit even if the frame is closed
- Give the frame a fixed size in pixels
  `public void setSize(int width, int height)`
- Resize the frame to fit the components tightly
  `public void pack()`

GUI example


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

public class GuiExample1 {
  public static void main(String[] args) {
    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(new Dimension(300, 100));
    frame.setTitle("A frame");
    JButton button1 = new JButton();
    button1.setText("I'm a button.");
    button1.setBackground(Color.BLUE);
    frame.add(button1);
    JButton button2 = new JButton();
    button2.setText("Click me!");
    button2.setBackground(Color.RED);
    frame.add(button2);
    frame.setVisible(true);
  }
}
```

- ❑ We defined two buttons, but only one is visible. Why?
- ❑ What happens when we click `button2`?

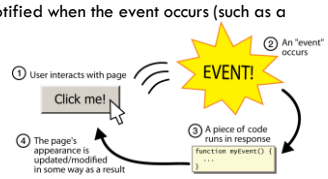
It's tedious… and there is more …

- Size and positioning
  - Preferred/minimum sizes, absolute/relative positioning
  - …
- Containers and layout
  - Flow layout – laying out components in a container
  - Border layout – `NORTH`, `SOUTH`, `EAST`, `WEST`, `CENTER`
  - Grid layout
  - …
- And more, lots more…
- …

GUI control structure

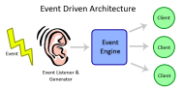
inversion-of-control

- *event*: An object representing a user's interaction with a GUI component
- *listener*: An object responding to events
- To handle an event, attach a listener to a component (such as a button)
- The listener will be notified when the event occurs (such as a button click)



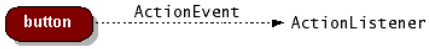
Event-driven programming

- A programming style where the overall flow of execution is dictated by events
- The program defines a set of listeners that wait for specific events
- As each event happens due to a user action, the program runs specific code
- The overall flow of execution is determined by the series of events that occur, not a pre-determined order
- The events invoke client code (through the listeners) without knowing which client code is invoked
  - The **invokes** relation (in part) no longer matches the **names** relation



Action events

- *action event*: An action on a GUI component
- The most common, general event type in Swing, caused by
  - button or menu clicks,
  - check box checking / unchecking,
  - pressing Enter in a text field, ...
- Represented by a class named `ActionEvent`
- Handled by objects that implement interface `ActionListener`



Implementing a listener

```
public class name implements ActionListener {
  public void actionPerformed(ActionEventevent) {
    code to handle the event;
  }
}
public void addActionListener(ActionListener al)
```

- Attaches the given listener to be notified of clicks and events that occur on this component
- `JButton` and other graphical components have this method

Event hierarchy

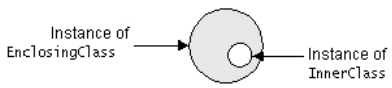
- `EventObject`
  - `AWTEvent (AWT)`
    - `ActionEvent`
    - `TextEvent`
    - `ComponentEvent`
      - `FocusEvent`
      - `WindowEvent`
      - `InputEvent`
        - `KeyEvent`
        - `MouseEvent`

```
import java.awt.event.*;
```

- `EventListener`
  - `AWTEventListener`
  - `ActionListener`
  - `TextListener`
  - `ComponentListener`
  - `FocusListener`
  - `WindowListener`
  - `KeyListener`
  - `MouseListener`

Nested classes

- *nested* class: A class defined inside of another class
  - Nested classes are hidden from other classes
  - Nested objects can access/modify the fields of their outer object
    - If necessary, can refer to outer object as `OuterClassName.this`
  - Only the outer class can see the nested class or make objects of it
- Event listeners are often defined as nested classes inside a GUI



GUI event example

```
public class MyGUI {
  private JFrame frame;
  private JButton stutter;
  private JTextField textfield;

  public MyGUI() {
    ...
    stutter.addActionListener(new StutterListener());
  }
  ...

  // When button is clicked, doubles the field's text
  private class StutterListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
      String text = textfield.getText();
      textfield.setText(text + text);
    }
  }
}
```

Mouse and keyboard events

- Low-level events – close to the hardware – to listen for and respond to mouse clicks/movements and keyboard entry/echoing

`MouseListener` interface

```
public interface MouseListener {
  public void mouseClicked(MouseEvent event);
  public void mouseEntered(MouseEvent event);
  public void mouseExited(MouseEvent event);
  public void mousePressed(MouseEvent event);
  public void mouseReleased(MouseEvent event);
}
```

- Most AWT/Swing components have this method
  `public void addMouseListener(MouseListener ml)`

Implementing listener

```
public class MyMouseListener implements
MouseListener {
  public void mouseClicked(MouseEvent event) {}
  public void mouseEntered(MouseEvent event) {}
  public void mouseExited(MouseEvent event)  {}
  public void mousePressed(MouseEvent event) {
    System.out.println("You pressed the button!");
  }
  public void mouseReleased(MouseEvent event) {}
}

// elsewhere,
myComponent.addMouseListener(new MyMouseListener());
```

- Tedious to define the empty method for the events you are **not** interested in

Adapter pattern to the rescue

- Provide an adapter class that connects to GUI components but exposes to us the interface we prefer – only a method or two
- *event adapter*: A class with empty implementations of all of a given listener interface's methods
  - Ex: `MouseAdapter`, `KeyAdapter`, `FocusAdapter`
  - Ex: To extend `MouseAdapter` only override methods you want to implement
    - Don't have to type in empty methods for the ones you don't want!

An abstract event adapter

```
// An empty implementation of all MouseListener methods
//   (from java.awt.event)
public abstract class MouseAdapter implements MouseListener {
  public void mousePressed(MouseEvent event) {}
  public void mouseReleased(MouseEvent event) {}
  public void mouseClicked(MouseEvent event) {}
  public void mouseEntered(MouseEvent event) {}
  public void mouseExited(MouseEvent event) {}
}
```

- Classes can extend `MouseAdapter` rather than implementing `MouseListener`
  - client gets the complete mouse listener interface it wants
  - implementer gets to write just the few mouse methods they want

```
public class MyMouseAdapter extends MouseAdapter {
  public void mousePressed(MouseEvent event) {
    System.out.println("You pressed the button!");
  }
}

// elsewhere…
myComponent.addMouseListener(new MyMouseAdapter());
```

Using MouseEvent

```
public class MyMouseAdapter extends MouseAdapter {
  public void mousePressed(MouseEvent event) {
    Object source = event.getSource();
    if (source == button && event.getX() < 10) {
      JOptionPane.showMessageDialog(null,
                 "You clicked the left edge!");
    }
  }
}
```

Mouse input listener

- The `MouseInputListener` interface and the `MouseInputAdapter` class ease the development of an object that listens to mouse clicks, movement, and/or wheel events

```
public class MyMouseInputAdapter extends
             MouseInputAdapter {
  public void mousePressed(MouseEvent event) {
    System.out.println("Mouse was pressed");
  }
  public void mouseDragged(MouseEvent event) {
    Point p = event.getPoint();
    System.out.println("Mouse is at " + p);
  }
}
…
MyMouseInputAdapter adapter = new
             MyMouseInputAdapter();
myPanel.addMouseListener(adapter);
myPanel.addMouseMotionListener(adapter);
```

Similar for keyboard events

```
public interface KeyListener {
  public void keyPressed(KeyEvent event);
  public void keyReleased(KeyEvent event);
  public void keyTyped(KeyEvent event);
}
…
// what key code was pressed? (one for almost every key)
public static int VK_A, VK_B, ..., VK_Z,
  VK_0, ... VK_9, VK_F1, ... VK_F10, VK_UP, VK_LEFT, ...,
// Were any modifier keys held down?
public static int CTRL_MASK, ALT_MASK, SHIFT_MASK
public char getKeyChar()
public int getKeyCode()     // use VK_* with this
public Object getSource()
public int getModifiers()  // use *_MASK with this
```

- Equivalent adapters, too

Focus: current target of keyboard input

- If a component doesn't have the focus, it will not receive events
- By default, most components don't receive focus
  - Buttons, text fields, and some others default to on
- `JComponent` methods for focus
  - `public void setFocusable(boolean b)`
    - Sets whether this component can receive keyboard focus
  - `public void requestFocus()`
    - Asks for this component to be given the current keyboard focus
- `FocusListener` (focus gained or lost), `focusAdapter`, also available

Other events

- Window events (closed, opened, iconified, ...)
- Change events (state changed in a `JSlider`, …)
- Component events (component hidden, resized, shown, …)
- `JList/JTree` select events
- Document events (for text fields)
- …

Next steps

- Assignment 3: due Sunday October 30, 11:59PM
- Lectures
  - W (Midterm review, including example questions)
- Upcoming: Friday 10/28, in class midterm – open book, open note, closed neighbor, closed electronic devices