Lecture 5
Java Swing, Layout Managers, Inner Classes, Listeners
Manu Kumar
sneaker@stanford.edu

HW#1: Pencil Me In Status!?

- Assigned Work Reminder:
  - HW #1: Pencil Me In
    - Due before midnight Wednesday, July 9th, 2003
    - You do have three floating late days, but use wisely!

- Reminder: Use office hours!
  - Questions on theory from class
  - Questions or clarifications on HW requirements
  - Development environment issues
  - Design issues

Random tips and pointers
- "Deprecated"
  - java.util.Date has lots of methods which are deprecated
  - Instead it references the "Calendar" class
- "Abstract"
  - java.util.Calendar is abstract!
  - The "concrete implementation" is actually in java.util.GregorianCalendar
- You do no need to do too much date arithmetic
  - But you do need to figure out how to use the API
  - The Java API is your friend. Use it well.

Summary
- Last Time
  - OOP/Inheritance
    - Pop-down rule
    - Constructors
    - instanceOf
    - Grad example
  - Abstract superclasses
    - Account example
    - Java Interfaces
    - Moodable example
  - Drawing in Java started
    - FirstFrame example
- Lots of Stuff!
  - Warning/Reminder: The summer quarter courses move fast!

Handouts
- 2 Handout for today!
  - #12: Inner Classes
  - #13: Listeners

Today
- Continue with Drawing in Java
  - Java Swing classes
    - JComponent
    - PaintComponent
    - Graphics Object
    - My Component Example
  - Layout Managers
    - Flow, Box and Border
    - Nesting
    - Layout Example
  - Inner Classes
  - Anonymous Inner Classes (maybe)
  - Listener model (maybe)
    - Button Listener Example
• Last time
  – FirstFrame example
  • Subclass JFrame
  • Get content pane
  • Set layout manager
  • Add components
    – Instantiate JLabel
    – Instantiate JButton
  • Pack
  • Set close behavior
  • Set visible

FirstFrame example

FirstFrame Code: getting started

// FirstFrame.java
/*
Demonstrates bringing up a frame with some labels.
*/
import java.awt.*;
import javax.swing.*;
import java.util.*;
import java.awt.event.*;
public class FirstFrame extends JFrame {
  public FirstFrame(String title) {
    super(title); // superclass ctor takes frame title
    // Get content pane -- contents of the window
    JComponent content = (JComponent) getContentPane();
    
    // Set to use the "flow" layout
    content.setLayout(new FlowLayout());
    // Background color is a property of all components --
    // set it to white
    content.setBackground(Color.lightGray);
    // Use add() to install components
    content.add(new JLabel("Hello World.");
    content.add(new JLabel("Another Label.");
    content.add(new JLabel("Klaatu Barada Nikto!");
    content.add(new JButton("Ok");
  }
  public static void main(String[] args) {
    new FirstFrame("First Frame");
  }
}

FirstFrame Code: adding components

FirstFrame Code: finishing touch

// Force the frame to size/layout its components
pack();
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// Java 1.3 or later
setVisible(true); // make it show up on screen

public static void main(String[] args) {
  new FirstFrame("First Frame");
}
Component Location/Size

• Each JComponent has its own coordinate system
  – (0,0) is in the top left corner
  – x grows to the right
  – Y grows to the left

• Bounds
  – Upper left corner (0.0)
  – component.getWidth()
  – component.getHeight()

• Local coordinate system
  – Does not change as the component is moved

Parent container
  – "parent" is the container the component is in
  – Parent is itself a component

"Location" of a component
  – The position of its upper left corner in the coordinate system of its parent

PreferredSize
  – Used by Layout Manager to determine the size of the component
  – setPreferredSize()
  – Can also use set minimum and maximum size to be considered by the layout manager

Layout Manager
  – Looks at the preferred size of all components and tries to do the best possible layout
  – Assigns final size and location
  • Use setPreferredSize before calling pack()
  • Hardly ever call setSize()

Size and Location messages
  – getWidth(), getHeight(), getSize(), getLocation(), getBounds()

Geometry methods
  • Mostly inherited from java.awt.Component
  • Constructor
    – Constructs a component with initial size zero
  • Methods
    – int getWidth(), getHeight()
    – Dimension getSize()
    – int getX(), getY()
    – Point getLocation()
    – getPreferredSize()
    – Rectangle getBounds()
    – boolean contains(x, y), boolean contains(Point p)
    – setBounds(x, y, width, height), setBounds(Rectangle)
    – getParent()

OOP GUI Drawing Theory

• Subclass JComponent
  • Override paintComponent()
    – Draw within the bound of the component
    – Install your components in a window/container
  • Remember:
    – Objects are anthropomorphic (like a person)
      • So we tell them how to do something (draw themselves)
      • Then send a message asking them to do the action (draw itself)

paintComponent(Graphics g)
  • Notification that is sent to a JComponent when it should draw itself
  • Override to provide custom drawing code
  • Call getWidth() etc to get geometry information
    – Do not hardcode!
  • Do no need to erase
    – Erased before paintComponent is called
  • Call super.paintComponent() for more complex cases
    – Often subclass JPanel instead
```java
public void paintComponent(Graphics g) {
    // not necessary for simple cases
    // super.paintComponent(g);
    int width = getWidth();
    int height = getHeight();
    // draw a rect around the bounds of the component
    // -1 since drawRect overhangs by one
    g.drawRect(0, 0, width-1, height-1);
    // draw a line from upper-left, to lower-right
    g.drawLine(0, 0, width-1, height-1);
}
```
/**
 * Draws a sort of face -- a rect at the bounds, two eyes, and a rect mouth. Draws a string "yo" string near the bottom.

 Typical paint component:
 -see how big you are
 -draw within your bounds
 -don't need to erase first -- canvas already erased
 */
 public void paintComponent(Graphics g) {
   // Could use this to get a sense of when drawing happens
   Toolkit.getDefaultToolkit().beep();
   int width = getWidth();
   int height = getHeight();

   // Draw a red rect at our bounds
   g.setColor(Color.red);
   g.drawRect(0, 0, width-1, height-1); // -1 for drawRect

   // eyes 1/3 from top, 1/3 from each side
   int eyeY = height/3;
   int left = width/3;
   int right = 2*width/3;
   int radius = width/15;

   // Draw two eyes
   g.setColor(Color.yellow);
   g.fillOval(left-radius, eyeY-radius, radius*2, radius*2);
   g.fillOval(right-radius, eyeY-radius, radius*2, radius*2);

   // Draw a little mouth from 1/4 to 3/4
   g.setColor(Color.lightGray);
   g.fillRect(width/4, 3*height/4, width/2, height/10);

   // Draw a string at 20, 20
   g.setColor(Color.black);
   g.drawString("yo!", 20, 20);
}

// Note: earlier examples subclassed off JFrame, and set things up in its ctor. In this case,
// we are just a client of JFrame, and send it messages like getContentPane() and pack().
// Both of these approaches are reasonable.

// Get the content area of the frame
JComponent content = (JComponent) frame.getContentPane();
content.setBackground(Color.white);
content.setLayout(new BoxLayout(content, BoxLayout.Y_AXIS));

// add a few components
content.add(new MyComponent(120, 80));
content.add(new MyComponent(120, 120));
content.add(new MyComponent(120, 140));

// Layout manager packs things to fit into the minimum window
frame.pack();

// frame.setSize(300, 200); // alternative to pack()
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);

// Theory
- Similar to HTML -- policy, not position
- Do not set explicit pixel sizes or positions of things
- Layout Managers know the intent (policy)
- Layout Managers apply the intent to figure out the correct size on the fly

// Advantages
- Platform independence
- Different platforms have different size fonts
- Resizing of windows
- Internationalization
- Adjust based on changing language

// Disadvantage
- Can sometimes be frustrating if it doesn’t do what you want!

Visual Hierarchy

- Components are placed inside other components
  - Resulting “hierarchy”
- Frames/Windows usually outermost components
- Constructed at run-time
  - JPanel which contains a JButton and several JLabels

Visual Hierarchy vs. Class Hierarchy

- Class hierarchy is a compile time hierarchy enforced by the compiler
- Visual hierarchy is how components are nested inside each other

Visual Hierarchy Example

- JFrame
  - JPanel (Smiley)
    - 8 Ovals
  - JPanel (ButtonPad)
    - 4 JButtons
  - JButton
  - JButton

FlowLayout

- Simplest
- Arranges components
  - Left to right
  - Top to Bottom
- Alignment options
  - RIGHT
  - LEFT
  - CENTER
  - LEADING
  - TRAILING

BoxLayout

- Aligns components in a line
  - Horizontally or vertically
- Can install a box layout into a JComponent
  - comp.setLayout(new BoxLayout(comp, BoxLayout.Y_AXIS))
- Or, create a “Box” Component
  - Box.createVerticalBox()
  - Box.createHorizontalBox()
  - Box.createVerticalStruts to create spacers between boxes
- See API documentation!

BoxLayout Example
BorderLayout

- Versatile layout
  - Can build very complex layouts by nesting BorderLayouts
- Main content in the “center”
  - Resize space allocated primarily to center
- Decorate borders on either side
  - North, South, East, West
- Takes second parameter to determine location
  - border.add(comp, BorderLayout.CENTER);

Nested JPanel

- JPanel is a simple component
  - Used to aggregate other components
    - Put multiple components in a JPanel using a given layout
    - Can then position the JPanel within another layout as if it were a complex component
  - To control the size of the elements in a panel we can use setPreferredSize
- Examples
  - Group label with a control
  - Set the layout of a vertical box and put lots of buttons in it and put it in the EAST of a BorderLayout

Layout Example
Inner Class

- A class definition inside a class
- Generally used as a private utility class which does not need to be seen by others classes
- Operates as a sub-part of the outer class
- It can have constructors, instance variables and methods, just like a regular class

Inner Class access

- Outer and inner classes can access each other state!
  - Even if private!
  - Stylistically, acceptable as they are both from a common code base
- Inner class always created inside a containing class (outer class)
  - It always has a pointer to the outer object
    - (Classname.this, example: Outer.this)
  - Can access instance variables automatically
- Use inner class when there is a natural need to access the variables of the outer class
  - Otherwise use a nested class (coming up!)

Nested Class

- Like an inner class
  - But does not have a pointer to the outer object
  - Does not have automatic access to the ivars of the outer object
- Users the static keyword
Inner/Nested Example

- Each inner object is created in the context of a single, "owning", outer object.
  - At runtime, the inner object has a pointer to its outer object which allows access to the outer object.
- Each inner object can access the ivars/methods of its outer object.
  - Can refer to the outer object using its classname as "Outer.this".
- The inner/outer classes can access each other's ivars and methods, even if they are "private".
  - Stylistically, the inner/outer classes operate as a single class that is superficially divided into two.

Inner/Nested Example Code

```java
public class Outer {
    private int a;
    private void increment() {
        a++;
    }
    private class Inner extends Object {
        private int b;
        private Inner(int initB) {
            b = initB;
        }
        private void demo() {
            // access our own ivar
            System.out.println("b: "+ b);
            // access the ivar of our outer object
            System.out.println("a: "+ a);
            // message send can also go to the outer object
            increment();
            "Outer.this refers to the outer object, so could say Outer.this.a or Outer.this.increment()
        }
    }
    public void test() {
        a = 10;
        Inner i1 = new Inner(1);
        Inner i2 = new Inner(2);
        i1.demo();
        i2.demo();
        Nested n = new Nested();
        n.demo();
    }
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

Inner/Nested Example Output

Output:

```
   b: 1
   a: 10
   b: 2
   a: 11
```
Anonymous Inner Classes

- Do not have a name
- Do not have a constructor
  - Relies on the default constructor of the superclass
- Does not have access to local stack variables (parameters to a method)
  - Unless they are declared final
- **Example**
  - Class Outer, Anonymous Inner class subclassed off of a class called Superclass

Anonymous Class Example

```java
public class Outer {
    int ivar;

    public Superclass method() {
        int sum; // ordinary stack var
        final int temp = ivar + 1; // stack var, but declared final (constant)
        Superclass s = new Superclass() {
            private int x = 0;
            public void foo() {
                x++; // x of inner class
                ivar++; // ivar of outer class
                bar(); // inherited from Superclass
                x = sum; // cannot see sum
                x = temp; // this works, since temp is final
            }
        };
        return(s); // later on, someone can send s.foo()
    }
...
```

final var trick

- Inner classes can see ivars of outer objects
- Inner classes **cannot** see stack variables (parameters)
- However
  - Inner classes can see “final” stack variables
- **Why**
  - Inlining of finals by the compiler
  - Declare stack variables as final to communicate their value to an anonymous inner class
- Outer.this os the pointer to the outer object

Controls and Listeners

- **Theory**
  - **Source**
    - Buttons, controls etc.
  - **Listener**
    - An Object that wants to know when the control is operated
  - **Notification Message**
    - A message sent from the source to the listener as a notification that the event has occurred
- Essentially: registering callbacks
Source-Listener Interaction

Listener Interface

- ActionListener Interface
  - Objects that would like to listen to a JButton must implement ActionListener

```java
public interface ActionListener extends EventListener {
    // Invoked when an action occurs.
    // *
    public void actionPerformed(ActionEvent e);
}
```

Notification Prototype

- The message prototype defined in the ActionListener Interface
  - The message the button sends
- ActionEvent parameter includes extra info
  - A pointer to the source object (e.getSource())
  - When the event happened
  - Any modifier keys held down

```java
public void actionPerformed(ActionEvent e);
```

source.addXXX(listener)

- To setup the listener relationship, the listener must register with the source
  - Example: button.addActionListener(listener)
- The listener must implement the ActionListener interface
  - It must respond to the message that the button will send

Using a Button and a Listener #1

- Component implements ActionListener
  - The component could implement the ActionListener interface directly
  - Register “this” as the listener object

```java
class MyComponent extends JComponent implements ActionListener {
    ...
    // in the JComponent ctor
    button.addActionListener(this);
}
```
• Create an inner class
  – Create a MyListener inner class which implements ActionListener
  – Create a new MyListener object
  – Add it via button.addXXX(listener)

// in the JComponent ctor
ActionListener listener = new MyActionListener();
button.addActionListener(listener);

Anonymous Inner class
• Most common method!
• Create an Anonymous inner Class that implements the interface
  – Can be created on the fly inside the method!

button = new JButton("Beep");
panel.add(button);
button.addActionListener(
  new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      Toolkit.getDefaultToolkit().beep();
    }
  });

ButtonListener Example Code
// ListenerFrame.java
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
import java.awt.event.*;
/**
 * Demonstrates bringing up a frame with a couple of buttons in it.
 * Demonstrates using anonymous inner class listener.
 */
public class ListenerFrame extends JFrame {
  private JLabel label;

  public ListenerFrame() {
    super("ListenerFrame");
    JComponent content = (JComponent) getContentPane();
    content.setLayout(new FlowLayout());
    JButton button = new JButton("Beep!");
    content.add(button);
    // ----
    // 1. Create an inner class subclass of ActionListener
    ActionListener listener = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        Toolkit.getDefaultToolkit().beep();
      }
    };
    // ----
    // 2. Add the listener to the button
    button.addActionListener(listener);
    // ----
    // 3. Create a listener in 1 step...
    // Create a little panel to hold a button and a label
    JPanel panel = new JPanel();
   (panel.add(button);
    JButton button2 = new JButton("Yay!");
    label = new JLabel("Woo Hoo!!!");
    panel.add(button2);
    panel.add(label);
// This listener adds a "!" to the label.
button2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // note: we have access to "label" of outer class
        // we do not have access to local vars like 'panel',
        // unless they are declared final.
        String text = label.getText();
        label.setText(text + "!");
    }
    pack();
    setVisible(true);
});

• JCheckBox
  – Uses ActionListener, like JButton
  – Responds to boolean isSelected() to see if it is currently checked

• JSlider
  – Component with min/max/current values
  – Users StateChangedListener interface
    • Notification is stateChanged(ChangeEvent e)
    • e.getSource() to get a pointer to the source
  – Responds to int getValue() to get current value

• Listener strategy
  – Our approach so far
  – Event based

• Polling strategy
  – Do not listen to the control
  – Check the value when you choose
  – Often fraught with problems, but may have an appropriate use in some cases