Lecture VII
JAVA SWING GUI TUTORIAL

These notes are based on the excellent book, "Core Java, Vol 1" by Horstmann and Cornell, chapter 7, graphics programming.

Introduction to AWT and Swing. AWT relies on "peer-based" rendering to achieve platform independence. But subtle difference in platforms resulted in inconsistent look-and-feel, and platform-dependent bugs. Swing avoids these problems by using a non-peer-based approach, with the result it may be slower than AWT. To recover the look-and-feel of each platform (Windows, Motif, etc), it allows programs to specify the look-and-feel. It also has a new look-and-feel, called "Metal". Note that AWT is not deprecated as a result of Swing.

Terminology:
Component: a user interface element that occupies screen space. E.g., button, text field, scrollbar.
Container: screen area or component that can hold other components. E.g., window, panel.
Event Detector: (non standard terminology?) I guess most components detects events and generates a corresponding event object. This is sent to the registered "listeners" for this event(component?).

§1. First Step: JFrame

The following gives the simplest standalone application involving Java GUI:

```
// file: EmptyFrame.java
// Adapted from Core Java, vol.1, by Horstmann & Cornell

import javax.swing.*;

class MyFrame extends JFrame {
  public MyFrame() {
setTitle("My Empty Frame");
setSize(300,200); // default size is 0,0
setLocation(10,200); // default is 0,0 (top left corner)
  }

  public static void main(String[] args) {
    JFrame f = new MyFrame();
    f.show();
  }
}
```

You can compile and run this program, but it does not do anything useful. It shows an empty window with the title "My Empty Frame".

A top-level window is a "frame". The AWT library has a peer-based class called Frame. In Swing, this is called JFrame. Indeed, most of the AWT components (Button, Panel, etc) has corresponding Swing counterparts named by prepending a "J" (JButton, JPanel, etc). JFrame is one of the few Swing components that are not drawn on a canvas. A JFrame is a "container" meaning it can contain other components such as bottons and text fields.

Question: what is the relation between f.show() and f.setVisible(true)? Ans: equivalent.

§2. Second Step: WindowListener

The above program is only hidden when you click the window close button. To truly kill the program, you need to type "CNTL-C". Our next program called EmptyFrame1.java will fix this problem.

This brings us to the GUI interaction model. When you click the window close button, it generates a window closing event. But some object has to be a "listener" for this event, and to act upon it. The Java model requires a **WindowListener** object for events generated by the frame. WindowListener is an interface with 7 methods to handle events of various kinds ("window closing event" is the one of interest here). When a window event occurs, the GUI model will ask the frame to handle the event using one of these 7 methods. Suppose you have implement WindowListener with a class "Terminator" which closes your window properly. Now, all you do is register an instance of Terminator:

```
class MyFrame extends JFrame {
  public MyFrame() {
    addWindowListener( new Terminator());
    ...
  }
  ...
}
```

But it is tedious to write a class "Terminator" to implement WindowListener when most of these 7 methods turn out to be null. So AWT provides a default implementation called **WindowAdapter** (found in java.awt.event.*) where all these 7 methods are null! But you can just extend this class and write any non-null methods to override the default:

```
class Terminator extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
}
```

Here is our actual code:

```
// file: EmptyFrame1.java

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

class EmptyFrame1 extends JFrame {

  // Constructor:
```

```
  public EmptyFrame1() {
setTitle("My Closeable Frame");
setSize(300,200); // default size is 0,0
setLocation(10,200); // default is 0,0 (top left corner)

// Window Listeners
addWindowListener(new WindowAdapter() {
   public void windowClosing(WindowEvent e) {
   System.exit(0);
   } //windowClosing
} );
  }

  public static void main(String[] args) {
    JFrame f = new EmptyFrame1();
    f.show();
  } //main
} //class EmptyFrame1
```

Note that we did not declare the terminator class; instead we use an anonymous class: `new WindowAdapter() { ... }` . Remark: sometimes, you may also need to call dispose() before System.exit(0), to return any system resources. But dispose alone without System.exit(0) is not enough.

§3. Third Step: Adding a Panel

We can next add panels to frames. The program called MyPanel.java illustrates adding a panel. There are 2 steps:

First, you need to define your own "MyPanel" class, which should extend the JPanel class. The main method you need to define in MyPanel is the "paintComponent" method, overriding the default method in JPanel.

Second, you need to add an instance of MyPanel to the JFrame. Not just the JFrame, but to a specific layer of the JFrame. A JFrame has several layers, but the main one for adding components is called "content pane". We need to get this pane:

```
Container contentPane = frame.getContentPane();
```

Then add various components to it. In the present example, we add a JPanel:

```
contentPane.add( new MyPanel());
```

§4. Fourth Step: Fonts in Panels

We next address the issue of fonts. Font families have several attributes:

- Font name. E.g. Helvetica, Times Roman
- Font style. E.g., PLAIN, BOLD, ITALIC
- Font size. E.g., 12 Point

To construct a Font object, do

```
Font helv14b = new Font("Helvetica", Font.BOLD, 14);
```

To use the font, call the setFont() method in the graphics object g: `g.setFont(helvb14);` You can also specify font styles such as `Font.BOLD + Font.ITALIC`. Use `getAvailableFontFamilyNames` of GraphicsEnvironment class to determine the fonts you can use. Instead of Font names, AWT defines 5 "logical font names":

```
SansSerif, Serif, Monospaced, Dialog, DialogInput
```

which are always available.

These concepts are illustrated below in our elaborated `paintComponent` method. The goal is ostensibly to print "Hello" in bold and "World!" in bold-italic fonts. To do this, we need to get the **FontMetrics** object which has methods to measure the length and height of a string, say.

```
/***********************************************************
         * @file: TextPanel.java
 * @source: adapted from Horstmann and Cornell, Core Java
 * @history: Visualization Course, Spring 2003, Chee Yap
 ***********************************************************/

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

/***********************************************************
         * TextPanel Class (with main method)
 ***********************************************************/

class TextPanel extends JPanel {
  // override the paintComponent method
  // THE MAIN DEMO OF THIS EXAMPLE:

  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Font f = new Font("SansSerif", Font.BOLD, 14);
    Font fi = new Font("SansSerif", Font.BOLD + Font.ITALIC, 14);
    FontMetrics fm = g.getFontMetrics(f);
    FontMetrics fim = g.getFontMetrics(fi);
    int cx = 75; int cy = 100;
    g.setFont(f);
    g.drawString("Hello, ", cx, cy);
```

```
    cx += fm.stringWidth("Hello, ");
    g.setFont(fi);
    g.drawString("World!", cx, cy);
  } //paintComponent

  //=============================================
  ///////////// main ///////////////////////////

  public static void main(String[] args) {
    JFrame f = new MyFrame("My Hello World Frame");
    f.show();
  } //main

} //class TextPanel

/**************************************************************
          MyFrame Class
 *************************************************************/

class MyFrame extends JFrame {
  public MyFrame(String s) {
// Frame Parameters
setTitle(s);
setSize(300,200); // default size is 0,0
setLocation(10,200); // default is 0,0 (top left corner)

// Window Listeners
addWindowListener(new WindowAdapter() {
  public void windowClosing(WindowEvent e) {
System.exit(0);
  } //windowClosing
}); //addWindowLister

// Add Panels
Container contentPane = getContentPane();
contentPane.add(new TextPanel());

  } //constructor MyFrame
} //class MyFrame
```

NOTE: The java.awt.FontMetrics.* class also has methods to get other properties of the font: its ascent, descent, leading, height, etc.

§5. Fifth Step: Basic Graphics

In "DrawFrame.java", we do basic 2D-graphics. We use the following primitives: drawXXX where XXX = Polygon, Arc, Line. We also use the "fillXXX" versions. Important point: in java.awt, you need to use a "canvas" to draw. In "Swing", you draw on any kind of panel.

More Basic Graphics In "DrawFrame1.java", we further look at the primitives drawXXX where

```
XXX = Rect, RoundRect, Oval.
```

We also replace "drawXXX" by "fillXXX".

§6. 6th Step: Basic Event Handling

This is shown in ButtonFrame.java demo.

We consider the class "Event", which are all derived from java.util.EventObject. Examples of events are "button pushed", "key pushed", "mouse moved". Some subclasses of Event are **ActionEvents** and **WindowEvents**.

To understand events, you need to consider two types of interfaces, and their relationship: **Event Detectors** and **Event Listeners**. The former is set up to detect the occurence of certain types of event, and it sends notices to the listeners. It is the listeners that take the appropriate action.

Examples of Event Detectors are windows or buttons. In this first event demo, we use buttons, as they generate the simplest kind of events. Buttons detects only one type of event – called ActionEvents. In contrast, there are seven kinds of WindowEvents. For buttons, the "ActionListener" is appropriate. But in order for the event objects to know which listener object to send the events to, we need to do three things:

(1) Implement the listener interface using ANY reasonable class. In our example, the class will be an extension of JPanel. To implement the ActionListener, you need to supply the method **actionPerformed(ActionEvent)** (the only method of this interface). The class for implementing Actionlistener here is "MyPanel":

```
public class MyPanel extends JPanel
implements ActionListener {
  public void actionPerformed(ActionEvent e){
    // reaction to button click goes here
    ...
  } // actionPerformed
} // class MyPanel
```

What is the action "..." above? This is explained below.

(2) Create a listener object:

```
Listener lis = new MyPanel();
```

(3) register this object with the event detector.

```
button.addActionListener(lis); // button is the event detector;
```

The general form for registering listener objects is:

```
<eventDetector>.add<EventType>Listener(<listenerObject>);
```

In our present demo, we will have two buttons (redButton and blueButton) in a panel. When redButton is pressed, the background of the panel changes to Red, and similarly when the blueButton is pressed. Thus, these buttons serve as event detectors. To use buttons, we need to create them:

```
private JButton redButton;
redButton = new JButton("RED"); // "RED" is label on button
```

In addition to (or instead of) "RED", we can supply an image file:

```
redButton = new JButton(new ImageIcon("RED.gif");
```

Next, you add the buttons to a panel (called ButtonPanel here). We also register the listener object with the buttons – but the listener object will be "this" (i.e., current object)!

```
class ButtonPanel extends JPanel {
  // members:
  private JButton redButton;
  private JButton blueButton;
  // constructors:
  public ButtonPanel() {

    // create buttons
    redButton = new JButton("RED");
    blueButton = new JButton("BLUE");

    // add buttons to current panel
    add(redButton);  // add button to current panel
    add(blueButton); // add button to current panel

    // register the current panel as listener for the buttons
    redButton.addActionListener(this);
    blueButton.addActionListener(this);

  } // ButtonPanel constructor
} // ButtonPanel class
```

We now return the details needed in the "actionPerformed(ActionEvent)" method from the ActionListener interface. First, you need to find out which Button caused this event. There are 2 ways to find out. First, the getSource() method from EventObject can be used:

```
Color color = getBackground();  // color will be set
Object source = e.getSource();
if (source == redButton) color = Color.red
else if (source == blueButton) color = Color.blue
setBackground(color);
repaint();  // when do we need this??
```

The second method, specific to ActionEvents, is to use the getActionCommand() method, which returns a "command string", which defaults to the button label. Thus,

```
String com = e.getActionCommand();
if (com.equals("RED")) ...; // "RED" is the label of redButton
else if (com.equals("BLUE")) ...;
```

But the command string need not be the label of the button. That can be independently set:

```
redButton.setActionCommand("RED ACTION");
```

The ActionListener interface is also used when (a) when an item is selected from a list box with a double click, (b) when a menu item is selected (c) when an enter key is clicked in the text field (d) when a specific time has elapsed for a "Timer" component.

In our present example of the red and blue buttons, the ActionListener interface is implemented by MyPanel for a good reason: the action of changing the background of the panel to red or blue ought to reside with the panel!

§7. 7th Step: Window Events

(NO DEMO PROGRAM HERE! FIX) We now consider the JFrame as an event detector. A JFrame is basically synonymous with a "window" and it detect Window Events. We need to implement the "WindowListener" interface to be registered with the JFrame. This interface has 7 methods:

```
public void windowClosed(WindowEvent e)
public void windowIconified(WindowEvent e)
public void windowOpened(WindowEvent e)
public void windowClosing(WindowEvent e)
public void windowDeiconified(WindowEvent e)
public void windowActivated(WindowEvent e)
public void windowDeactivated(WindowEvent e)
```

Recall that we already had a brief introduction to this interface, when we extended the "WindowAdapter" class to provide default empty implementations for all but the windowClosing method, which we implement by a call to "System.exit(0)". In Step 2 above, the extension of WindowAdapter was called Terminator. In general, all the AWT listener interfaces with more than one method comes with such an adapter class. Finally, we we create an instance of the Terminator class and register it with the JFrame using the "addWindowListener(WindowListener wl)" method:

```
class MyFrame extends JFrame {
  // Constructor:
  public MyFrame() {
    addWindowListener(new Terminator());
    ...
```

```
  } // MyFrame Constructor
  ...
} // MyFrame Class
```

We can even make the ”Terminator” anonymous, as an inner class, as follows:

```
class MyFrame extends JFrame {
  // Constructor:
  public MyFrame() {
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      } // windowClosing
    }); // WindowAdapter
    ...
  } // MyFrame Constructor
  ...
} // MyFrame Class
```

In general, you can create an anonymous class by the construct

```
new XXXclass() {... override methods, etc... }
```

where `XXXclass` is the class name.

Inner classes are also extremely useful, because the inner class can automatically get access to all the methods and fields of its parent class. In Component design, you often want a new class to subclass two different classes. But Java does not allow this. You get around this restriction by subclassing a subclass – an inner class is one way of doing this.

An example where you want to program the ”windowDeactivated”, ”windowActivated”, ”windowIconified” and ”windowDeiconified” is when your window displays an animation. You would want to stop or start the animations when these events occur.

§8. 8th Step: Event Classes and Listener Interfaces

Now that we have seen two types of events (ActionEvents and WindowEvents), let us overview the general picture. Here is the event hierarchy:

```
 | ActionEvent*     | ContainerEv.*
 | AdjustmentEv.*   | FocusEvent*   | KeyEvent*
   EventObject <-- AWT Event <--| ComponentEv.* <--| InputEvent <--|
 | ItemEvent*     | PaintEvent    | MouseEv.*
 | TextEvent*     | WindowEvent*
```

In this hierarchy, only 10 event classes, those indicated with asterisks (*) are actually passed to listeners. The 10 event classes are classified into 4 ”semantic” events and 6 ”low-level” events. Intuitively, semantic events correspond to what the user intends (e.g., button click), while low-level events correspond to physical events.

Semantic Events
1) ActionEvent: button click, menu selection, selecting item in list, typing ENTER in text field
2) AdjustmentEvent: the user adjusted a scroll bar.
3) ItemEvent: the use made a selection from a set of checkbox or list items
4) TextEvent: the contents of a text field or area were changed.

Low-Level Events
1) ComponentEvent: component is resized, moved, shown, hidden. It is the base class for all low-level events.
2) KeyEvent: a key was pressed or released.
3) MouseEvent: the mouse button was depressed, released, moved, dragged.
4) FocusEvent: a component got focus, lost focus.
5) WindowEvent: window was (de)activated, (de)iconified, or closed.
6) ContainerEvent: a component has been added or removed. Usually, you don't have to worry about this class of event, as these events are (usually) not generated dynamically, but in your program.

All low-level events is derived from ComponentEvent. They all have a method "getComponent" (it is similar to the "getSource" method but the result is already cast as a component).

There are 11 listener interfaces in java.awt.event:

```
ActionListener,  AdjustmentListener,ComponentListener*
ContainerListener*,KeyListener*,MouseListener*
MouseMotionListener*,TextListener,FocusListener*
ItemListener,WindowListener*.
```

The listeners with asterisks (*) have a corresponding adaptor class implementing it because they each have more than one method. There is a 1-1 correspondence between listeners and event types, with one exception: MouseEvents are sent to both MouseListeners and MouseMotionListener. The split into two types of listeners for MouseEvents is done for efficiency – so we can ignore an entire class of mouse events (such as mouse motion which can generate frequent events).

§9. 9th Step: Focus Event

(NO DEMO PROGRAM HERE) (A) In Java, a component has the "focus" if it can receive key strokes. E.g., a text field has the focus when the cursor mark becomes visible, ready to receive key strokes. When a button has "focus", you can click it by pressing the space bar. (B) Only one component can have the focus at any moment. The user can choose the component to have focus by a mouse click in the component (in some system, just having the mouse cursor over a component is sufficient). The TAB key also moves the focus to the "next" component, and thus allows you to cycle over all "focusable" components. Some components like labels or panels are not "focusable" by default. You can make any component "focusable" or not by overriding the "isFocusTraversable" method to return TRUE or FALSE. You can use the "requestFocus" method to move the focus to any specific component at run time, or you can use "transferFocus" method to move to the next component. The notion of "next" component can be changed. (C) The FocusListener interface has 2 methods: focusGained and focusLost. Each takes the "FocusEvent" object as parameter. Two useful methods for implementing this interface are "getComponent" and "isTemporary". The latter returns TRUE if the focus lost is temporary and will automatically be restored.

§10. 10th Step: KeyBoard Events and Sketch Demo

The "keyPressed" and "keyReleased" methods of the KeyListener interface handles raw keystrokes. However, another method "keyTyped" combines the response to these two types of events, and returns the characters actually typed. Java distinguished between "characters" and "virtual key codes". The latter are indicated by the prefix of "VK_" such as "VK_A" and "VK_SHIFT". These 3 methods are best illustrated with an example:

Suppose a user types an lower case "a". There are only 3 events:
(a) Pressed A key (keyPressed called for VK_A)
(b) Typed "a" (keyTyped called for character "a")
(c) Released A key (keyReleased called for VK_A)

Now, suppose the user types an upper case "A". There are 5 events:
(a) Pressed the SHIFT key (keyPressed called for VK_SHIFT)
(b) Pressed A key (keyPressed called for VK_A)
(c) Typed "A" (keyTyped called for character "A")
(d) Released A key (keyReleased called for VK_A)
(e) Released SHIFT key (keyReleased called for VK_SHIFT)

To work with keyPressed and keyReleased methods, you need to check the "key code".

```
public void keyPressed(KeyEvent e) {
  int keyCode = e.getKeyCode();
  ...
}
```

The key code (defined in the KeyEvent class) is one of the following constants:

```
VK_A VK_B ... VK_Z
VK_0 ... VK_9
VK_COMMA VK_PERIOD ... etc
```

Instead of tracking the key codes in the case of combination strokes, the following methods which returns a Boolean value are useful: "KeyEvent.isShiftDown()", "KeyEvent.isControlDown()", "KeyEvent.isAltDown()" and "KeyEvent.isMetaDown()".

To work with keyTyped method, you can call the "getKeyChar" method.

The following demo is a "Etch-A-Sketch" toy where you move a pen up, down, left, right with cursor keys or h, j, k, l. Holding down the SHIFT key at the same time will move the pen by larger increments.

§11. 11th Step: Mouse Events and Mouse Demo

Some mouse events such as clicking on buttons and menus are handled internally by the various components and translated automatically into the appropriate semantic event (e.g, handled by the actionPerformed or itemStateChanged methods). But to draw with a mouse, we need to trap mouse events.

When the user clicks a mouse button, three kinds od MouseEvents are generated. The corresponding three MouseListener methods are: "mousePressed", "mouseReleased" and "mouseClicked". The last method generates only one event for each pressed-released combination. To obtain the position of the mouse when the events occur, use the methods MouseEvent.getX() and MouseEvent.getY(). To distinguish between single and double (and even triple) clicks, use the MouseEvent.getClickCount() method.

SIMPLIFY the example in the book (how about drawing a line to the current mouse click?) But I already have this under the "rubber line" example.

§12. 12th Step: Action Interface

(I) WHAT ARE ACTIONS? Java even model allows us to choose any class as listener for its events. So far, we have let each class (i.e., "this") be its own listener. For bigger examples, we want to separate the responsibilites of detecting from listening/responding. We need an independent notion of "action" (="responding to action"). This is logically sensible, since the same action may be needed for different events. For example, suppose the action is to "set background color" (to red/blue, etc). This action may be initiated in 3 ways:
(a) click on a color button (as in example above)
(b) selection of a color in a set-background menu
(c) press of a key (B=blue, R=red, etc).

(II) METHODS OF ACTION INTERFACE: Java provides the "Action" interface with these methods:
(1) void actionPerformed(ActionEvent e) – performs the action corresponding to event e
(2) void setEnabled(boolean b) – this turns the action on or off
(3) boolean isEnabled() – checks if the action is on
(4) void putValue(String key, Object val) – store a value under a key (String type). There are 2 standard keys: Action.NAME (name of action) and Action.SMALL_ICON (icons for action). E.g., Action.putValue(Action.SMALL_ICON, new ImageIcon("red.gif"));
(5) Object getValue(String key) – retrieve the value stored under key
(6) void addPropertyChangeListener(PropertyChangeListener listener) – Add a "Changelistener" object to our current list. Menus and toolbars are examples of "ChangeListeners", as these components must be notified when a property of an action that they are responsible for changes.
(7) void removePropertyChangeListener(PropertyChangeListener listener) – Similar to method in (6), but this one is to remove a "ChangeListener" from current list.
Actually, "Action" extends "ActionListener" and method (1) above was the ONLY method in the ActionListener interface (cf. the ButtonFrame.java demo above). This method Thus, an Action object can be used whenever an ActionListener object is expected.

(III) MenuAction.java: IMPLEMENTING and USING AN ACTION INTERFACE. There are three steps:
STEP 1: Define a class implementing the Action interface
STEP 2: Instance the Action class
STEP 3: Associate the action instance with components
STEP 4: Add the components to the windowing system

STEP 1: As usual, there is a default implementation of the Action interface, called "AbstractAction". You can adapt from this class, and only "actionPerformed" method needs to be explicitly programed by you. Usually, you also want to provide a constructor to set the values stored under various keys, and your class will want a member variable "target" to remember the component where the action is to be performed (recall that the Action object need not be the component itself, after our decoupling of "event generator" from "event listener"). Here is an implementation of the action "set background color":

```
class BackgroundColorAction extends AbstractAction {
  //members:
    private Component target;  // where you want the action done!

  //constructor:
  public BackgroundColorAction(
String name, Icon i, Color c, Component comp) {
    putValue(Action.NAME, name);
    putValue(Action.SMALL\_ICON, i);
    putValue("Color", c);
    target = comp;
  } // constructor

  //methods:
  public void actionPerformed(ActionEvent e) {
    Color c = (Color)getValue("Color");
    target.setBackground(c);
    target.repaint();
  }// actionPerformed method
}// BackgroundColorAction class
```

STEP 2: we need to instance the class:

```
Action redAction = new BackgroundColorAction(
"Red", new ImageIcon("red.gif"), Color.red, panel);
```

STEP 3: associate the action with components or their instances: The following associates "redAction" with a component instance. The component illustrated here is a JButton instance.

```
JButton redButton = new JButton("Red");
redButton.addActionListener(redAction);
```

Alternatively, we associate any given "Action" with an entire class of components. For JButtons, we create button class that comes with an action:

```
class ActionButton extends JButton {
  public ActionButton(Action a) {
    setText((String)a.getValue(Action.NAME));
    Icon icon = (Icon)a.getValue(Action.SMALL\_ICON);
    if (icon != null) setIcon(icon)
    addActionListener(a);
  }// ActionButton constructor
}// ActionButton class

// instance it!
redButton = new ActionButton(redAction);
```

NOTE: If we introduce "ActionButtons" then STEPS 2 and 3 should be interchanged!

STEP 4: Now add ActionButtons to a menu, then to the menu bar:

```
  // create the menu of ActionButtons:
JMenu m = new JMenu("Color");
m.add(redAction);
m.add(blueAction);

// add menu to menubar
JMenuBar mbar = new JMenuBar();
mbar.add(m);
setJMenubar(mbar);
```

END OF LECTURE