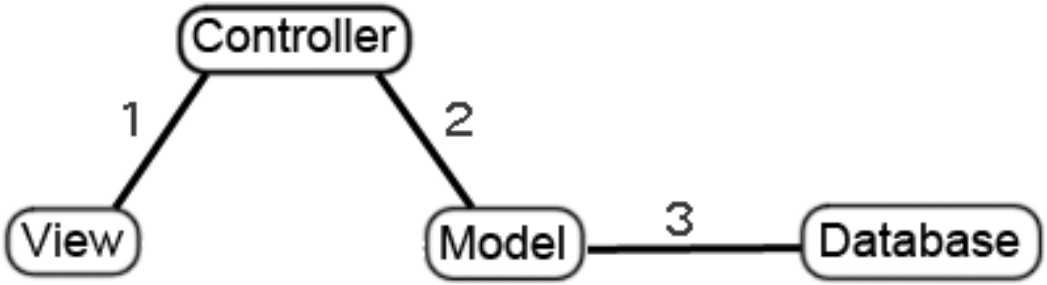
Unit - 5

Swing Fundamentals

Swing Controls

Origins and Philosophy of Swing

- Swing is a collection of classes and interfaces that offer a rich set of visual components (push buttons, text fields, scroll bar, check boxes, menus) using which we can build GUI applications.
- Swing overcomes the limitations of AWT (Abstract Window Toolkit) such as reliance on platform-specific native windows.
- Introduced in 1997, Swing offers two key features:
  - Lightweight components: Components are written entirely in Java. The components can work in a consistent manner across all platforms.
  - Pluggable look and feel: It is possible to “plug-in” a new look and feel for any component without creating any side effects in the code that uses the component.
    - Based on MVC (Model-View-Controller) architecture

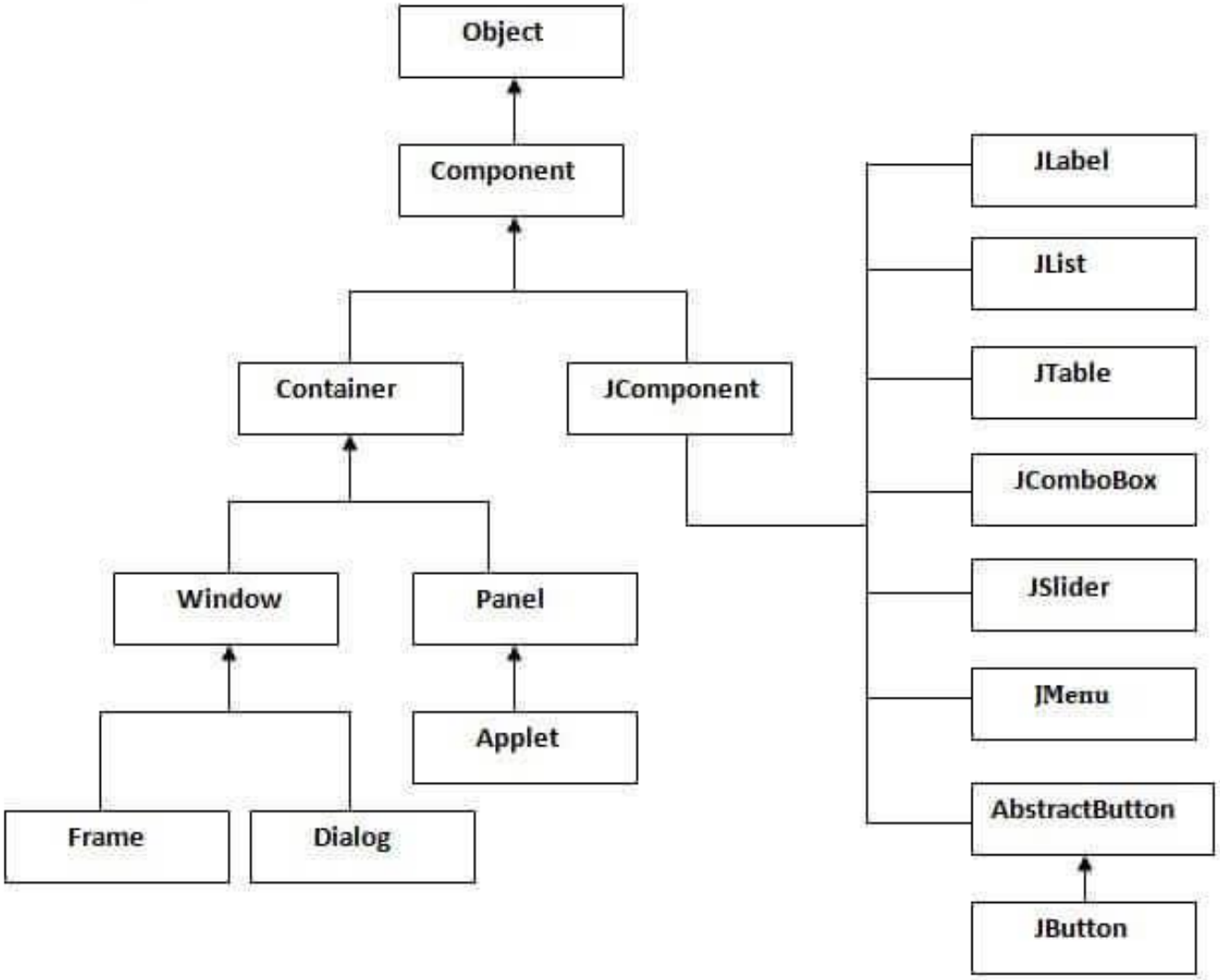


1. All actions begin in the view through events generated by the user. The controller provides *listeners* for the events.
2. The controller interacts with the model by either requesting information from the data source based on user-generated events, or by modifying the data based on these events.
3. The model provides the programming interface which the controller must use to access the database.

Difference between AWT and Swing

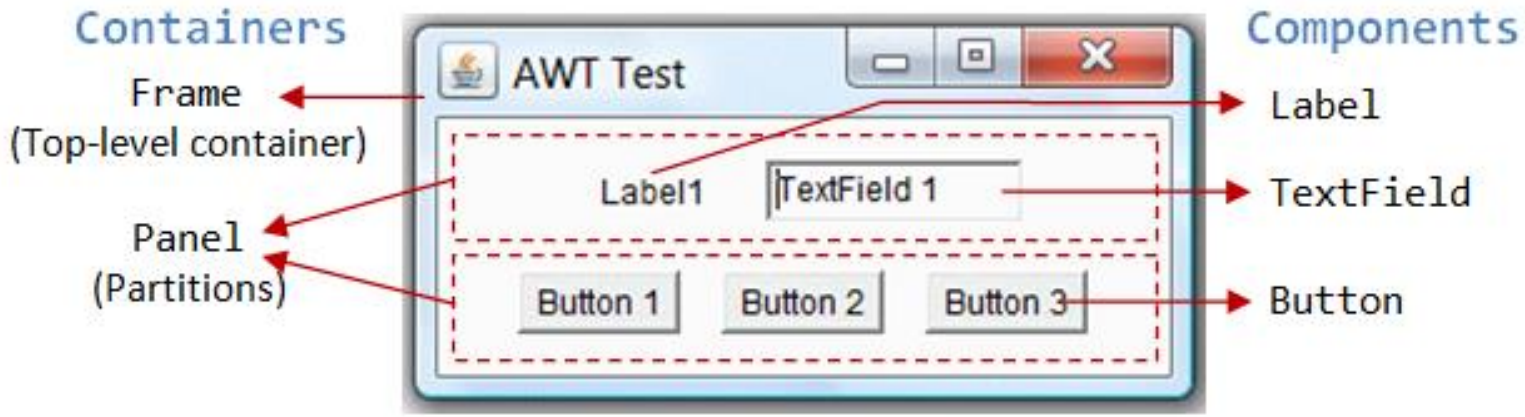
AWT	Swing
Platform Dependent	Platform Independent
Does not follow MVC	Follows MVC
Lesser components	More powerful components
Does not support pluggable look and feel	Supports pluggable look and feel
Heavyweight	Lightweight

Hierarchy of Java Swing classes



Components and Containers

- A component is an independent visual control, such as push button.
- A container holds a group of components.
  - A container is a special type of component that is designed to hold other components.
  - All Swing GUIs will have at least one container.



Components

- Swing components are derived from JComponent class. JComponent inherits AWT classes Container and Component.
- All Swing components are represented by classes defined within the package javax.swing.

JApplet	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayeredPane
JList	JMenu	JMenuBar	JMenuItem
JOptionPane	JPanel	JPasswordField	JPopupMenu
JProgressBar	JRadioButton	JRadioButtonMenuItem	JRootPane
JScrollBar	JScrollPane	JSeparator	JSlider
JSpinner	JSplitPane	JTabbedPane	JTable
JTextArea	JTextField	JTextPane	JTogglebutton
JToolBar	JToolTip	JTree	JViewport
JWindow			

Containers

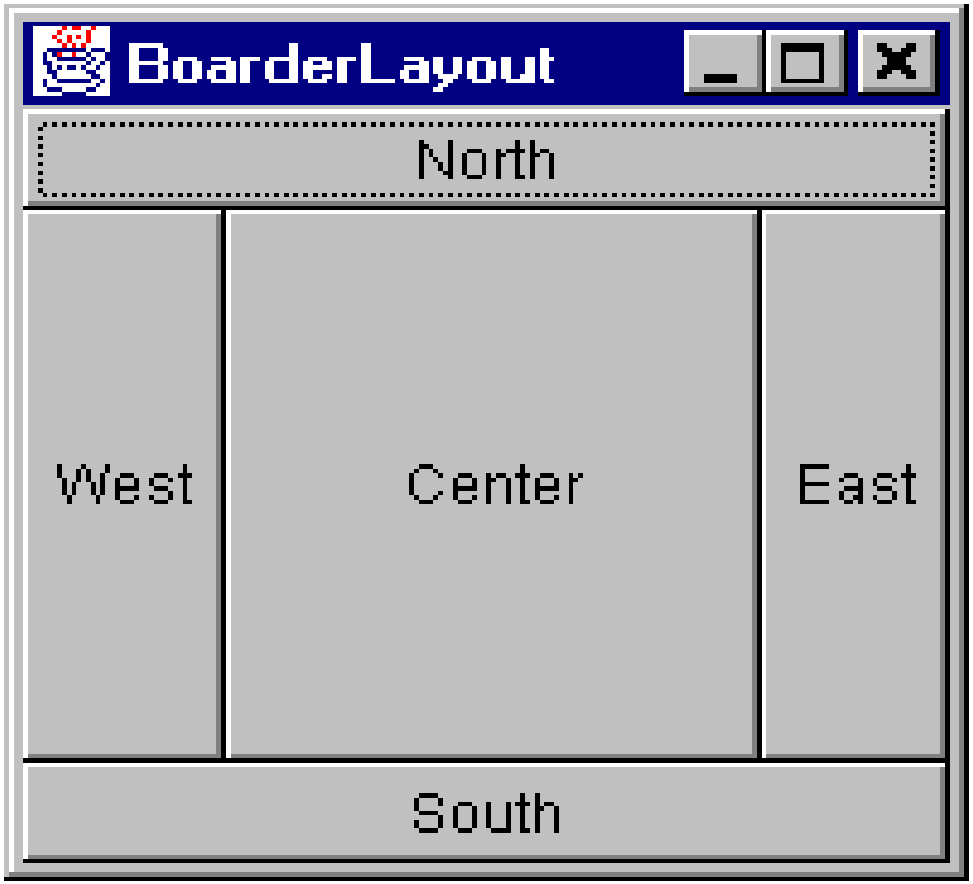
- Swing defines two types of containers.
- Top-level containers:
  - JFrame, JApplet, JWindow and JDialog.
  - These don't inherit JComponent. They inherit Component and Container.
  - A top-level container cannot be contained within an other container.
  - Most commonly used container for applications is JFrame.
- Lightweight containers:
  - JPanel, JRootPane
  - Used to organize and manage groups of related components.
  - Can be contained within another container.

Layout Managers

- The layout manager controls the position of components within a container. Some layout managers are:
  - FlowLayout: A simple layout that positions components left-to-right, top-to-bottom
  - BorderLayout: Positions components within the center or the borders of the controller.
  - GridLayout: Lays out components within a grid.
  - BoxLayout: Lays out components vertically or horizontally within a box.
  - GridBagLayout: Lays out different size components within a flexible grid.

BorderLayout

- It has four narrow, fixed-width components at the edges and one large area in the center. The four sides are referred to as north, south, east, and west. The middle area is called the center.
- **BorderLayout** defines the following constants that specify the regions:
  - BorderLayout.CENTER
  - BorderLayout.SOUTH
  - BorderLayout.EAST
  - BorderLayout.WEST
  - BorderLayout.NORTH
- When adding components, you will use these constants with the **add( )**, which is defined by **Container**:
  - void add(Component *compObj*, Object *region*)
  - Here, *compObj* is the component to be added, and *region* specifies where the component will be added.


BoarderLayout
North
West
Center
East
South

FlowLayout

- By default, components are laid out line-by-line beginning at the upper-left corner.
  - In all cases, when a line is filled, layout advances to the next line. A small space is left between each component, above and below, as well as left and right.

A first simple Swing Program

```
import javax.swing.*;
public class SwingDemo1 {
   SwingDemo1() {
      JFrame jf = new JFrame("My first Swing application!"); //Top-level container
      jf.setSize(400, 100); //Container size
      jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      JLabel jlb = new JLabel("Click here! Nothing happens!!"); //Create a JLabel
      jf.add(jlb); //Add the label to content pane
      jf.setVisible(true);
   }
   public static void main(String[] args) {
      SwingUtilities.invokeLater(new Runnable() {
         public void run() {
            new SwingDemo1(); //Create GUI on event dispatching thread
         }
      });
   }
}
```

Explanation…

- setSize(int width, int height); //in pixels
- By default, when a top-level window is closed, the window is removed from the screen, but the application is not terminated. The *setDefaultCloseOperation(int* what*)* is used to terminate the entire application.
  - what can be
    - EXIT_ON_CLOSE
    - DISPOSE_ON_CLOSE
    - HIDE_ON_CLOSE
    - DO_NOTHING_ON_CLOSE
- All top-level containers have a content pane in which components are stored.
- By default, the content pane associated with a JFrame uses a border layout.
- By default, a JFrame is invisible, so **setVisible(true)** must be called to show it.

Explanation …

```
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
            new SwingDemo1();
    }
});
```

- To enable the GUI code to be created on the event-dispatching thread, we use the invokeLater(Runnable obj) method.
  - Allows us to post a "job" to Swing, which will then run on the event dispatch thread at its next convenience.
- Here an unnamed new Runnable object is created that will have its run() method called by the event-dispatching thread.

Event Handling

- Many Swing controls respond to user input, and the events generated by those interactions need to be handled.
  - Button click, Key press, item selection in a list etc.
- Event handling mechanism is based on an approach called event delegation model.
  - A source generates an event and sends it to one or more listeners.
  - A listener registers with a source in order to receive an event notification.
  - Listener simply waits until it receives an event.
  - Once an event arrives, the listener processes the event and then returns.
- This model has the advantage of separating the application logic that processes events from the user interface logic that generates the events.
  - A UI element is able to “delegate” the processing of an event to a separate piece of code.

Events

- An event is an object that describes a state change in a source.
  - It can be generated as a consequence of user interacting with a control in a GUI or it can be generated under program control.

Event Sources

- An event source is an object that generates an event.
  - When a source generates an event, it must send that event to all registered listeners.
  - Listeners register with a source by calling an **add*type*Listener()** method on the event source object. Each type of event has its own registration method.
    - public void add*Type*Listener(*Type*Listener e)
      - *Type* is the name of the event and e is a reference to the event listener.
    - A method that receives key stroke event is called addKeyListener()
    - A method that registers a mouse motion listener is called addMouseMotionListener()
  - When an event occurs, all registered listeners are notified.
  - A source event must also provide a method that allows a listener to unregister an interest in a specific type of event.
    - public void remove*Type*Listener(*Type*Listener e)
  - The methods that add or remove listeners are provided by the source that generates events.

Event Listeners

- A listener is an object that is notified when an event occurs.
  - First, it must have registered with one or more sources to receive notifications about a specific type of event.
  - Second, it must implement a method to receive and process that event.
- The methods that receive and process events are defined in a set of interfaces.
  - For example, the ActionListener interface defines a method that receives a notification when an action, such as clicking a button, takes place.
  - Any object may receive and process this event if it provides an implementation of the ActionListener interface.

Using a Push Button

- One of the simplest and most commonly used Swing control.
- Swing push buttons can contain text, an image, or both.
  - JButton(String msg)
    - The msg parameter specifies the string that will be displayed inside the button
- When a push button is pressed, it generates an **ActionEvent**.
- The **JButton** provides two methods (inherited from **AbstractButton**), which are used to add or remove an action listener.
  - void **addActionListener**(ActionListener e)
  - void **removeActionListener**(ActionListener e)
  - Here e specifies an object that will receive event notification.
  - This object must be an instance of a class that implements the **ActionListener** interface.
  - The **ActionListener** interface defines only one method: **actionPerformed**()

- The method **actionPerformed**() will be called when a button is pressed.
- Using **ActionEvent** object passed to **actionPerformed**(), we can obtain many useful pieces of information.
  - The **getActionCommand**() returns a string that specifies the string displayed inside the button.
  - When using two or more buttons within the same application, the action command gives us an easy way to determine which button was pressed.

```
import java.awt.*; //contains FlowLayout class
import java.awt.event.*; //Defines ActionListener interface and ActionEvent class
import javax.swing.*;
public class SwingDemo2 implements ActionListener {
   JLabel jlb;
   SwingDemo2() {
      JFrame jf = new JFrame("A Button Example");
      jf.setLayout(new FlowLayout());
      jf.setSize(300, 100);
      jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      JButton b1 = new JButton("First");
      JButton b2 = new JButton("Second");
      b1.addActionListener(this);
      b2.addActionListener(this);
      jf.add(b1);
      jf.add(b2);
      jlb = new JLabel("Press a button.");
      jf.add(jlb);
      jf.setVisible(true);
   }
```

```
public void actionPerformed(ActionEvent  e) {
        if(e.getActionCommand().equals("First"))
            jlb.setText("First  button was clicked.");
        else
            jlb.setText("Second  button was clicked.");
    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new SwingDemo2();
            }
        });
    }
}
```

- java.awt package contains the FlowLayout class.
  - FlowLayout lays out components one “line” at a time, top to bottom. When one “line” is full, layout advances to the next “line.
- java.awt.event package defines the ActionListener interface and the ActionEvent class.
- SwingDemo2 class implements ActionListener. This means that SwingDemo2 objects can be used to receive action events.

Introducing JTextField

- JTextField (inherited from abstract class JTextComponent) from enables the user to enter a line of text.
  - JTextField(int *cols*)
    - Here *cols* specifies the width of the text field in columns. (Just the physical size; Not the number of characters that you may enter)
- When the use presses ENTER when entering text, an ActionEvent is generated.
  - To handle action events, we must implement the actionPerformed() method defined by the ActionListener interface.
- To obtain the string that is currently displayed in the text field, we use getText() method on the JTextField object.
- We can set the text in a JTextField using the method: setText(String text)

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class SwingDemo3 implements ActionListener {
   JTextField t1;
   JLabel jbl;
   SwingDemo3() {
      JFrame jf = new JFrame("A Text Field Example");
      jf.setLayout(new FlowLayout());
      jf.setSize(300, 100);
      jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      t1 = new JTextField(10);
      t1.addActionListener(this);
      jf.add(t1);
      jbl = new JLabel("");
      jf.add(jbl);
      jf.setVisible(true);
   }
```

```
public void actionPerformed(ActionEvent  e) {
        jbl.setText("Current  contents:" + t1.getText());
    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new SwingDemo3();
            }
        });
    }
}
```

- Like a Jbutton, a JTextField has an action command string associated with it.
  - By default, the action command is the current contents of the text field. However, we can set this to an action command of our choice by calling:
    - void setActionCommand(String cmd);
  - The string passed in cmd becomes the new action commnad. The text in the text field is unaffected.
  - The setActionCommand() is helpful when we have other controls in the same frame that also generate action events and we want to use the same event handler to process both events.
  - Setting the action command is one way we can tell them apart.

```
public class TwoTFDemo implements ActionListener {
   JTextField t1, t2;
   JLabel jbl;
   TwoTFDemo() {
      JFrame jf = new JFrame("Two Text Fields Example");
      jf.setLayout(new FlowLayout());
      jf.setSize(300, 100);
      jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      t1 = new JTextField(10);
      t2 = new JTextField(10);
      t1.setActionCommand("One");
      t2.setActionCommand("Two");
      t1.addActionListener(this);
      t2.addActionListener(this);
      jf.add(t1);
      jf.add(t2);
      jbl = new JLabel("");
      jf.add(jbl);
      jf.setVisible(true);
   }
```

```
 public void actionPerformed(ActionEvent  e) {
        if(e.getActionCommand().equals("One"))
           jbl.setText("ENTER  pressed in t1:" + t1.getText());
        else
           jbl.setText("ENTER  pressed in t2:" + t2.getText());
    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
           public void run() {
              new TwoTFDemo();
           }
        });
    }
}
```

Using Anonymous Inner Classes to Handle Events

- In the previous programs, the main class implemented the listener interface itself and all events are sent to an instance of that class.
- Other ways of handling events are
  - Implement separate listener classes (separate from the main class of the application) that handle different events.
  - Implement listeners through the use of anonymous inner classes.
- An inner class is a non-static class that is declared within another class.
- Anonymous inner classes are inner classes that don't have a name.
  - An instance of the anonymous class is simply generated "on the fly" as needed.

```
b1.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            showStatus("Button clicked!");
        }
});
```

```
public class AnonInnClassDemo {
   JLabel jlb;
   AnonInnClassDemo() {
      JFrame jf = new JFrame("A Button Example");
      jf.setLayout(new FlowLayout());
      jf.setSize(300, 100);
      jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      JButton b1 = new JButton("First");
      JButton b2 = new JButton("Second");
      b1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
               jlb.setText("First button was clicked.");
            }
      });
      b2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
               jlb.setText("Second button was clicked.");
            }
      });
      jf.add(b1);
      jf.add(b2);
      jlb = new JLabel("Press a button.");
      jf.add(jlb);
      jf.setVisible(true);
   }
```

```
public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new AnonInnClassDemo();
            }
        });
    }
}
```

Exploring Swing Component Classes

- We will be exploring the following Swing component classes.

Jbutton	JCheckBox	JComboBox
JLabel	JList	JRadioButton
JScrollPane	JTable	JTextField
JToggleButton	JTree	

JLabel and ImageIcon

- JLabel is a passive component; It does not respond to user input.
- Earlier, we used a JLabel to display a text message. However, it can also be used to display an icon (a graphics image) or both an icon and text.
- JLabel has the following constructors:
  - JLabel(String str, int align);
  - JLabel(Icon icon, int align);

- We can also use setIcon() and setText() methods to change the image and/or text on the label.

```
public class LabelIconDemo implements ActionListener {
    JLabel L1, jlb;
    JButton b1, b2;
    ImageIcon icon;
    LabelIconDemo() {
        JFrame jf = new JFrame("JLabel and ImageIcon Demo");
        jf.setLayout(new FlowLayout());
        jf.setSize(400, 400);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        L1 = new JLabel("Are you happy?");
        jf.add(L1);
        b1 = new JButton("Yes");
        b2 = new JButton("No");
        b1.addActionListener(this);
        b2.addActionListener(this);
        jf.add(b1);
        jf.add(b2);
        jlb = new JLabel("");
        jf.add(jlb);
        jf.setVisible(true);
    }
```

```
public void actionPerformed(ActionEvent e) {
        if(e.getActionCommand().equals("Yes")) {
            icon = new ImageIcon("happy.jpeg");
            jlb.setIcon(icon);
            jlb.setText("    Good");
        }
        else {
            icon = new ImageIcon("sad.jpg");
            jlb.setIcon(icon);
            jlb.setText("    Too Bad!");
        }
    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new LabelIconDemo();
            }
        });
    }
}
```

The Swing Buttons

- Java supports a variety of Swing button types.

JButton	A standard push button
JToggleButton	A two-state (on/off) button
JCheckBox	A standard check box
JRadioButton	A mutually exclusive check box

- All buttons are subclasses of AbstractButton class, which extends Jcomponent.

- Button events fall into three categories:
  - Action event: Generated when the user performs an action, such as clicking a button.
  - Item event: Generated when a button is selected or deselected.
  - Change event: Generated when the state of the button changes.

Handling Action Events

- An action event is generated when the user clicks a button.
- Action events are handled by classes that implement the ActionListener interface.
  - The interface specifies actionPerformed() method.
    - void actionPerformed(ActionEvent e)
  - The ActionEvent object passed to actionPerformed() helps us to identify which component has generated the event.
  - We can obtain the action command string for the component that generated the event by calling getActionCommand() on the ActionEvent object.
    - String getActionCommand()
    - void setActionCommnad(String cmd)
  - Another way to identify the component that generated an action event is by obtaining a reference to it by calling getSource() on the ActionEvent object.

Handling Item Events

- An item event occurs when an item, such as toggle button, check box, or radio button, is selected.
- Item events are represented by ItemEvent class.
  - Item events are handled by classes that implement the ItemListener interface.
  - This interface defines only one method:
    - void itemStateChanged(ItemEvent e)
  - The item event is received in e.

JToggleButton

- A useful variation on the push button is called a *toggle button*.
- A toggle button looks just like a push button, but it acts differently because it has two states: pushed and released.
  - That is, when you press a toggle button, it stays pressed rather than popping back up as a regular push button does.
  - When you press the toggle button a second time, it releases (pops up).
  - Therefore, each time a toggle button is pushed, it toggles between its two states.

- JToggleButton(String *str*)
  - This creates a toggle button that contains the text passed in *str*. By default, the button is in the off position.
- Like **JButton**, **JToggleButton** generates an action event each time it is pressed.
  - Unlike **JButton**, however, **JToggleButton** also generates an item event.
- When a **JToggleButton** is pressed in, it is selected. When it is popped out, it is deselected.
  - To handle item events, you must implement the **ItemListener** interface.
  - Each time an item event is generated, it is passed to the **itemStateChanged( )** method defined by **ItemListener**. Inside **itemStateChanged( )**, the **getItem( )** method can be called on the **ItemEvent** object to obtain a reference to the **JToggleButton** instance that generated the event.

```
public class JToggleButtonDemo {
   JLabel l1;
   JToggleButton jt1;
   JFrame jf;
   JToggleButtonDemo() {
      jf = new JFrame("Toggle Button Demo");
      jf.setSize(300, 100);
jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      jf.setLayout(new FlowLayout());
      jt1 = new JToggleButton("On/Off");
      l1 = new JLabel("Button is off");
      jt1.addItemListener(new ItemListener() {
         public void itemStateChanged(ItemEvent ie) {
            if(jt1.isSelected())
               l1.setText("Button is on.");
            else
               l1.setText("Button is off.");
         }
      });
      jf.add(jt1);
      jf.add(l1);
      jf.setVisible(true);
   }
```

```
 public static void main(String[] args) {
      SwingUtilities.invokeLater(new  Runnable() {
         public void run() {
            new JToggleButtonDemo();
         }
      });
   }
}.
```