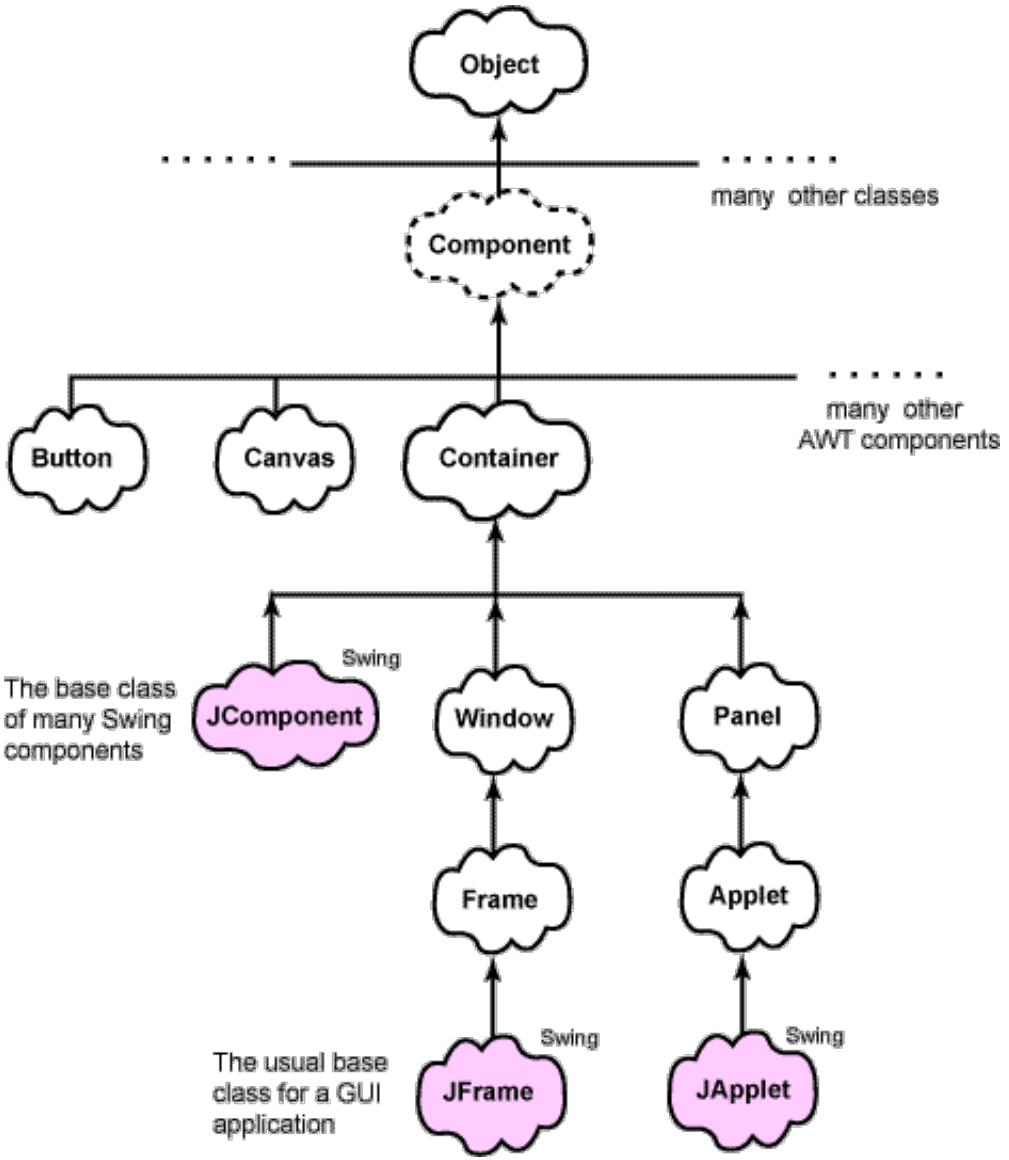
AWT hierarchy

Awt is heavy-weight bcz it uses components/resources of OS and also as it uses resources of OS, appearance of components changes with OS, i.e., button will look different on Windows , Linux etc.

AWT

- Java AWT (Abstract Window Toolkit) contains the fundamental classes used for developing GUI or window-based applications in java.
- AWT is heavyweight, use OS resource components directly. That is, appearance of component changes with OS version.
- The **java.awt** package provides classes for AWT api such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.
- The abstract **Component** class is the base class for the AWT.
- Many AWT classes are derived from it. These are the old AWT components that are no longer in use.
- Some examples of the AWT classes derived from Component are Button, Canvas, and Container.

AWT hierarchy



Button, text fields, scroll bars, etc are all components and in java AWT, there are classes for each component. In order to place each component on its correct position or on the screen, we need to add them to a container.

AWT hierarchy(2)

- The **Component** class
  - It is at the top of the AWT hierarchy.
  - Encapsulates all of the attributes of a visual component.
  - Defines over a hundred public methods for managing events, such as mouse and keyboard input, positioning and sizing the window, foreground color , background color and repainting
- The **Container** class
  - A subclass of Component
  - Contains other components like buttons, textfields, labels etc.
  - A container lays out (positioning) any components that it contains by using various layout managers

Container is basically a screen where the components are placed at their position. The classes that extend container class are known as container such as frame, panel, dialog, etc.

AWT hierarchy(3)

- The **Panel** class
  - A concrete subclass of Container
  - Doesn't contain title bar, border and menu bars.
  - It can have other components like button, textfield etc.
- The **window** class
  - Creates a top-level window
  - Frame subclass is used for this purpose
  - Have no borders and menu bars. Must use frame, dialog or another window for creating a window.

AWT hierarchy(4)

- The **Frame** class
  - A subclass of Window
  - Contains title bar and can have menu bars.
  - It can have other components like button, textfield etc.
- The **JComponent** class is derived from Container and is one of the base classes of Swing.
- The **JFrame** class is derived from the AWT Frame class. It is usually the main container for a GUI application.
- The **JApplet** class is derived from the AWT Applet class and is used for modern applets.

Difference between AWT and Swing ---

AWT	Swing
platform-dependent	independent
heavyweight	lightweight
provides less components than Swing.	provides more powerful components than AWT.
doesn't follow MVC.	follow MVC.

Swing

Why AWTs are no longer in use?

- Because the AWT components use native code resources, they are referred to as heavyweight.
- Problems with AWTs
  - A component might look, or even act, differently on different platforms (OS).
  - This threatened the overarching philosophy of Java: write once, run anywhere.
  - The use of heavyweight components caused some frustrating restrictions. For example, a heavyweight component is always rectangular and opaque.

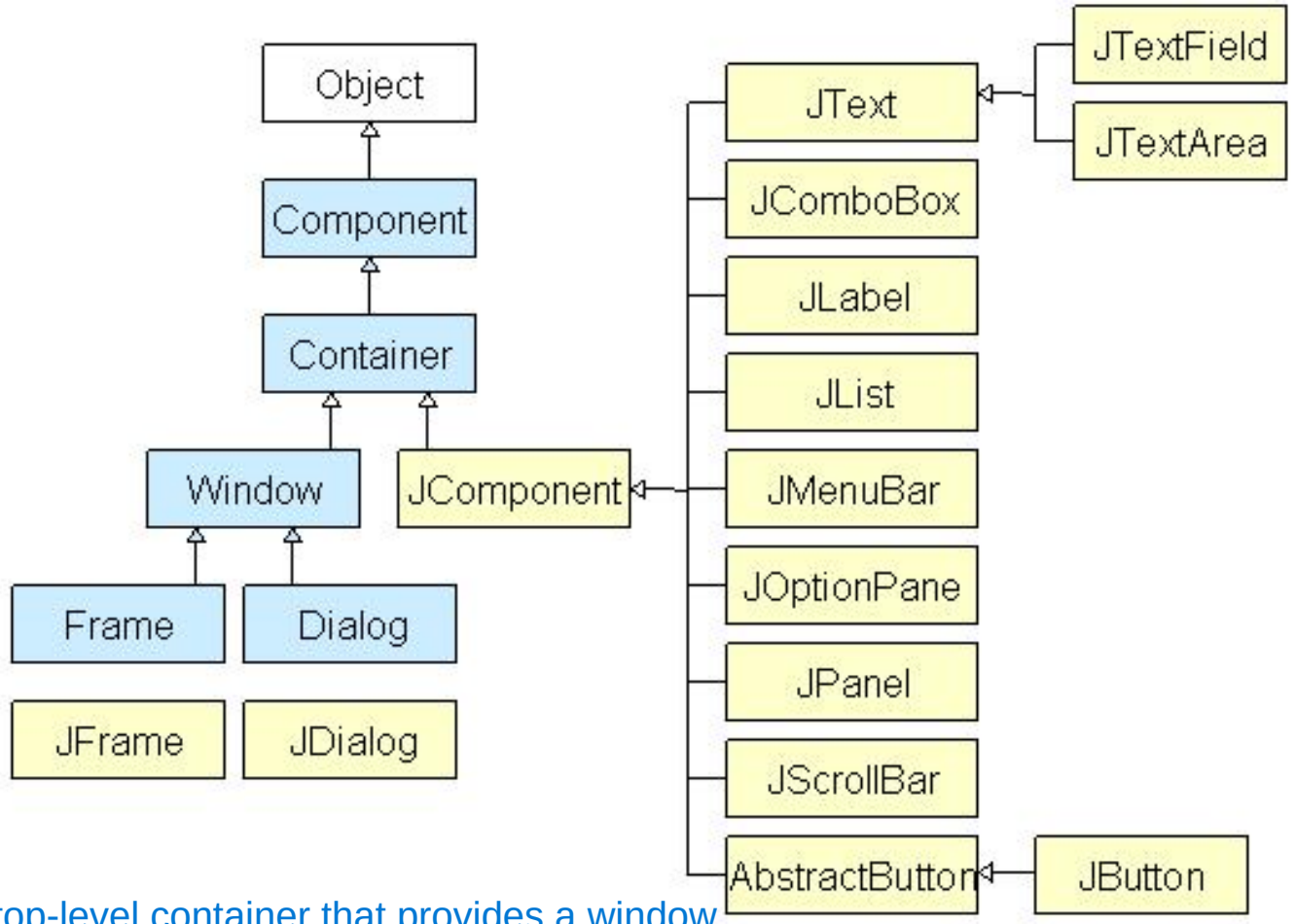
How Swing taken care these issues?

- Swing is built on the foundation of the AWT. This is why the AWT is still a crucial part of Java.
- Swing also uses the same event handling mechanism as the AWT
- Swing components are lightweight. This means that they are written entirely in Java and do not map directly to platform-specific peers (a few exceptions)
- Swing supports a pluggable look and feel (PLAF).

Swing Components and Containers

- A **component** is an independent visual control, such as a push button or slider.
- A **container** is a special type of component designed to holds a group of components.
- In order for a component to be displayed, it must be held within a container.
- Thus, all Swing GUIs will have at least one container.
- Swing can define a containment hierarchy, at the top of which must be a top-level container.

Components



JFrame is a top-level container that provides a window on the screen.
JFrame comes under Swing Component whereas Frame comes under AWT component.
JFrame inherits Frame class. It works like the main window where components like label, buttons are added to create GUI. It has more functionality than Frame.

Top-level container panes

- Each top-level container defines a set of panes.
- At the top of the hierarchy is an instance of **JRootPane**. It manages the other panes and the optional menu bar.
- The root pane: glass pane, content pane, and layered pane.
- **Glass pane**
  - The top-level pane sits above and completely covers all other panes.
  - By default, it is a transparent instance of **JPanel**.
  - Enables to manage mouse events that affect the entire container
  - to paint over any other component
- **Layered pane**
  - An instance of JLayeredPane
  - allows components to be given a depth value that determines which component overlays another

Top-level container panes(2)

- **Content pane**
- The layers of a container can be thought as transparent film that overlays the container.
- In Java Swing, the layer that is used to hold objects is called the **content pane**.
- The content pane is an object created by the Java run time environment.
- Objects are added to the content pane layer of the container.
- The getContentPane() method
  - It retrieves the content pane layer to add an object to it.
  - When you use getContentPane(), the content pane object then is substituted there so that you can apply a method to it.
  - A JFrame is the head component which is put together with other subcomponents.
  - For example, a JMenuBar is placed in another area next to the contentPane of a frame.

Layout Managers are used to arrange components in a particular manner. It facilitates us to control and size the components in GUI forms. It is an interface which is implemented by all classes of layout managers.

Layout Managers

- A layout manager is an instance of any class that implements the LayoutManager interface.
- The layout manager is set by the setLayout( ) method.
- If no call to setLayout( ) is made, then the default layout manager is used.
- On resizing a container, the layout manager is used to position each of the components within it.
- The setLayout( ) method:
  - **void setLayout(LayoutManager layoutObj)**
  - layoutObj is a reference to the desired layout manager
  - For manual layout, set layoutObj as null

Some Layout Managers Classes

Border Layout Group Layout
Flow Layout
Grid Layout
Box Layout
Card Layout GridBag Layout

- **FlowLayout**
  - Default layout manager
  - By default, orientation is left to right, top to bottom
- **FlowLayout constructors**
  - FlowLayout( )
  - FlowLayout(int how)
  - FlowLayout(int how, int horz, int vert)
  - Here, how is the alignment (FlowLayout.LEFT, FlowLayout.CENTER, FlowLayout.RIGHT, FlowLayout.LEADING, and FlowLayout.TRAILING)
  - horz and vert are the horizontal and vertical space between components respcetively

Some Layout Managers Classes(2)

- **GridLayout**
  - It lays out components in a two-dimensional grid.
  - Requires the number of rows and columns when instantiated.
- **GridLayout constructors:**
  - GridLayout( )
  - GridLayout(int numRows, int numColumns)
  - GridLayout(int numRows, int numColumns, int horz, int vert)

Some Layout Managers Classes(3)

doubt ?

- **GridBagLayout**
- We can specify the component's positions within cells inside a grid. It's components may not be of same size.
- It's a collection of small grids joined together.
  - GridBagLayout constructors: GridBagLayout( )
- One important method is **setConstraints( )**
  - void setConstraints(**Component** comp, **GridBagConstraints** cons)
  - Here, comp is the component for which the constraints specified by cons apply.
  - **GridBagConstraints** defines several fields that you can set to govern the size, placement, and spacing of a component

The Swing Packages

javax.swing	javax.swing.border	javax.swing.colorchooser
javax.swing.event	javax.swing.filechooser	javax.swing.plaf
javax.swing.plaf.basic	javax.swing.plaf.metal	javax.swing.plaf.multi
javax.swing.plaf.synth	javax.swing.table	javax.swing.text
javax.swing.text.html	javax.swing.text.html.parser	javax.swing.text.rtf
javax.swing.tree	javax.swing.undo	

A Simple Example

```
// A simple Swing application.
import javax.swing.*;
class SwingDemo {
        SwingDemo() {
                // Create a new JFrame container.
                JFrame jfrm = new JFrame("A Simple Swing Application");
                // Give the frame an initial size.
                jfrm.setSize(275, 100);
                // Terminate the program when the user closes the application.
                jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                // Create a text-based label.
                JLabel jlab = new JLabel(" Swing means powerful GUIs.");
                // Add the label to the content pane.
                jfrm.add(jlab);
                // Display the frame.
                jfrm.setVisible(true);
        }
        public static void main(String args[]) {
                // Create the frame on the event dispatching thread.
                SwingUtilities.invokeLater(new Runnable() {
                        public void run() {
                                new SwingDemo();
                         }
                });
        }
}
```

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs.
JAVA uses Delegation Event Model to handle the events.

Event Handling

- **Delegation event model**
  - Defines standard and consistent mechanisms to generate and process **events**.
  - A **source** generates an event and sends it to one or more **listeners**.
  - The listener simply waits until it **receives** an event.
  - Once an event is received, the listener **processes** the event and then returns
  - Listeners must **register** with a source in order to receive an event notification.
- **The advantage**
  - The application logic that processes events is cleanly separated from the user interface logic that generates those events.
  - A user interface element is able to “delegate” the processing of an event to a separate piece of code.
  - Notifications are sent only to listeners that want to receive them.
  - Eliminates this overhead of old approach of event handling

Events

- An event is an object that describes a state change in a source.
- A consequence of a person interacting with the elements in a GUI.
  - Pressing a button
  - Entering a character via the keyboard
  - Selecting an item in a list
  - Clicking the mouse and many more
- Events may also occur that are not directly caused by interactions with a user interface.
  - When a timer expires
  - A counter exceeds a value
  - A software or hardware failure occurs
  - An operation is completed.

callback Methods ?

Event Sources

- A source is an object that generates an event. May generate more than one type of event.
- A source must register listeners in order for the listeners to receive notifications about a specific type of event.
- Each type of event has its own registration method.
- The general form:
  - public void addTypeListener(TypeListener el)
- Example
  - AddKeyListener( ), addMouseMotionListener( ).
- When an event occurs, all registered listeners are notified and receive a copy of the event object via multicasting the event.
- In all cases, notifications are sent only to listeners that register to receive them.

Event Listeners

- General form of adding one and only one listener
  - public void addTypeListener(TypeListener el) throws java.util.TooManyListenersException
- Event is notified to the registered listener using unicasting of the event.
- A source must also provide a method that allows a listener to unregister an interest in a specific type of event.
  - public void removeTypeListener(TypeListener el) // Example removeKeyListener( ).

Events in Swing p1

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

class EventDemo {
        JLabel jlab;
        EventDemo() {
                JFrame jfrm = new JFrame("An Event Example");
                jfrm.setLayout(new FlowLayout()); // Specify FlowLayout for the layout manager.
                jfrm.setSize(220, 90);
                jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

                JButton jbtnAlpha = new JButton("Alpha");
                JButton jbtnBeta = new JButton("Beta");

                jbtnAlpha.addActionListener(new ActionListener() { // Add action listener for Alpha.
                        public void actionPerformed(ActionEvent ae) {
                                jlab.setText("Alpha was pressed.");
                        }
                });
                jbtnBeta.addActionListener(new ActionListener() { // Add action listener for Beta.
                        public void actionPerformed(ActionEvent ae) {
                                jlab.setText("Beta was pressed.");
                        }
                });
```

Events in Swing p2

```
jfrm.add(jbtnAlpha);
        jfrm.add(jbtnBeta);
        jlab = new JLabel("Press a button."); // Create a text-based label.

        jfrm.add(jlab);
        jfrm.setVisible(true);
    }
    public static void main(String args[]) {
        SwingUtilities.invokeLater(new Runnable() { // Create the frame on the event dispatching thread.
            public void run() {
                new EventDemo();
            }
        });
    }
}
```

JLabel and ImageIcon

object of JLabel is a component of placing text. It is used to display a single line read-only text.

- It creates a label
- JLabel can be used to display text and/or an icon.
- A passive component (no response to user input)
- Some JLabel constructors
  - JLabel(Icon icon)
  - JLabel(String str)
  - JLabel(String str, Icon icon, int align)
- ImageIcon class is the easy way to get an icon
  - ImageIcon(String filename)
- The icon and text associated with the label can be obtained by
  - Icon getIcon( )
  - String getText( )
- The icon and text associated with a label can be set by
  - void setIcon(Icon icon)
  - void setText(String str)

Example

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JLabelDemo" width=250 height=150>
</applet>
*/

public class JLabelDemo extends JApplet {
	public void init() {
		try {
			SwingUtilities.invokeAndWait(
				new Runnable() {
					public void run() {
						makeGUI();
					}
				}
			);
		} catch (Exception exc) {
			System.out.println("Can't create because of " + exc);
		}
	}
	private void makeGUI() {
		ImageIcon ii = new ImageIcon("france.gif");
		JLabel jl = new JLabel("France", ii, JLabel.CENTER);
		add(jl);
	}
}
```

JTextField

- JTextField allows you to edit one line of text.
- Derived from JtextComponent
- Provides the basic functionality common to Swing text components.
- JTextField uses the Document interface
- JTextField's constructors:
  - JTextField(int cols)
  - JTextField(String str, int cols)
  - JTextField(String str)
- str is the string of initialization, and cols is the no. of columns in the text field.
- Field is intially empty, if no string is specified
- By default, no. of columns fit the specified string.

JTextField(2)

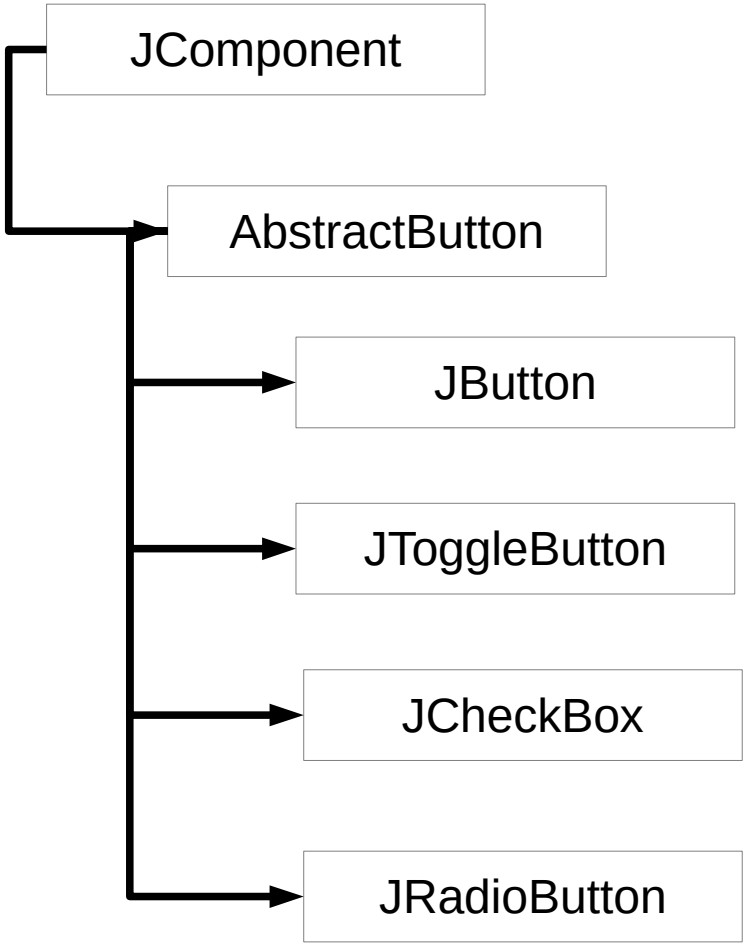
- JTextField generates events in response to user interaction.
  - Example:
  - ActionEvent is fired, when the user presses ENTER
  - CaretEvent is fired, when cursor changes position.
- The most common action is to obtain the string currently in the text field
  - using getText( ).

Example

```
// Demonstrate JTextField.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JTextFieldDemo" width=300
height=50>
</applet>
*/
public class JTextFieldDemo extends JApplet {
        JTextField jtf;
        public void init() {
                try {
                        SwingUtilities.invokeAndWait(
                            new Runnable() {
                                public void run() {
                                    makeGUI();
                                }
                            }
                        );
                } catch (Exception exc) {
                        System.out.println("Can't create
                        because of " + exc);
                }
        }
```

```
   private void makeGUI() {
          setLayout(new FlowLayout());
          jtf = new JTextField(15);
          add(jtf);
          jtf.addActionListener(new ActionListener() {
             public void actionPerformed(ActionEvent ae) {
                 showStatus(jtf.getText());
             }
          });
    }
}
```

Swing Buttons



- The button text can be read and written via:
  - String getText( )
  - void setText(String str)
- A button generates an action event when it is pressed

JButton

class used to create a labelled button.

- A push button
- Constructors
  - JButton(Icon icon)
  - JButton(String str)
  - JButton(String str, Icon icon)
- When the button is pressed, an ActionEvent is generated.
- Using the ActionEvent object passed to the actionPerformed( ) method of the registered ActionListener, the action command string can be obtained.
- We can set the action command by calling setActionCommand( ) on the button.
- We can obtain the action command by calling getActionCommand( ) on the event object.
  - String getActionCommand( )

Example : JButton

```
// Demonstrate an icon-based JButton.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JButtonDemo" width=250 height=450>
</applet>
*/
public class JButtonDemo extends Japplet implements ActionListener {
        JLabel jlab;
        public void init() {
                try {
                        SwingUtilities.invokeAndWait(
                                new Runnable() {
                                        public void run() {
                                                makeGUI();
                                        }
                                }
                        );
                } catch (Exception exc) {
                                System.out.println("Can't create because of " + exc);
                }
        }
```

Contd in next slide...

Example : JButton(2)

```
    private void makeGUI() {
        setLayout(new FlowLayout());
        ImageIcon france = new ImageIcon("france.gif");
        JButton jb = new JButton(france);
        jb.setActionCommand("France");
        jb.addActionListener(this);
        add(jb);
        ImageIcon germany = new ImageIcon("germany.gif");
        jb = new JButton(germany);
        jb.setActionCommand("Germany");
        jb.addActionListener(this);
        add(jb);
        ImageIcon italy = new ImageIcon("italy.gif");
        jb = new JButton(italy);
        jb.setActionCommand("Italy");
        jb.addActionListener(this);
        add(jb);
        ImageIcon japan = new ImageIcon("japan.gif");
        jb = new JButton(japan);
        jb.setActionCommand("Japan");
        jb.addActionListener(this);
        jlab = new JLabel("Choose a Flag");
        add(jlab);
    }

    // Handle button events.
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("You selected " + ae.getActionCommand());
    }
}
```

Check Boxes

- The JCheckBox class provides the functionality of a check box.
- Its immediate superclass is JToggleButton, which provides support for two-state buttons.
- JcheckBox defines several constructors: one is
  - JCheckBox(String str)
  - It creates a check box that has the text specified by str as a label.
- On selection or deselection (checked or unchecked) a check box, an **ItemEvent** is generated.
- A reference to the **JCheckBox** that generated the event by calling **getItem( )** on the ItemEvent passed to the itemStateChanged( ) method defined by ItemListener.
- To determine the selected state of a check box is to call **isSelected( )** on the JcheckBox instance.

Example: JCheckbox

```
// Demonstrate JCheckbox.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JCheckBoxDemo" width=270 height=50>
</applet>
*/
public class JCheckBoxDemo extends Japplet implements ItemListener {
        JLabel jlab;
        public void init() {
                try {
                        SwingUtilities.invokeAndWait(
                                new Runnable() {
                                        public void run() {
                                                makeGUI();
                                        }
                                }
                        );
                } catch (Exception exc) {
                        System.out.println("Can't create because of " + exc);
                }
        }

        private void makeGUI() {
                setLayout(new FlowLayout());
                JCheckBox cb = new JcheckBox("C"); // Add check boxes to the content pane.
                cb.addItemListener(this);
                add(cb);
                cb = new JCheckBox("C++");
                cb.addItemListener(this);
                add(cb);
```

Contd. In next slide...

Example: JCheckbox(2)

```
        cb = new JCheckBox("Java");
        cb.addItemListener(this);
        add(cb);
        cb = new JCheckBox("Perl");
        cb.addItemListener(this);
        add(cb);

        jlab = new JLabel("Select languages");  // Create the label and add it to the content pane.
        add(jlab);
    }

    // Handle item events for the check boxes.
    public void itemStateChanged(ItemEvent ie) {
        JCheckBox cb = (JCheckBox)ie.getItem();
        if(cb.isSelected())
            jlab.setText(cb.getText() + " is selected");
        else
            jlab.setText(cb.getText() + " is cleared");
    }
}
```

The JTabbedPane class is used to switch between a group of components by clicking on a tab with a given title or icon.

JTabbedPane

JPanel - simply a container in which an application can add/attach other components.

- Encapsulates a tabbed pane
- It manages a set of components by linking them with tabs.
- JTabbedPane uses the SingleSelectionModel model.
- Tabs are added by calling addTab( ):
  - void addTab(String name, Component comp)
  - Here, name is the name for the tab, and comp is the component that should be added to the tab.
  - Often, a JPanel is added as component
- Generic Steps
  - Create an instance of JTabbedPane.
  - Add each tab by calling addTab( ).
  - Add the tabbed pane to the content pane

Example: JtabbedPane p1

```
// Demonstrate JTabbedPane.
import javax.swing.*;
/*
<applet code="JTabbedPaneDemo" width=400 height=100>
</applet>
*/
public class JTabbedPaneDemo extends JApplet {
        public void init() {
                try {
                        SwingUtilities.invokeAndWait(
                                new Runnable() {
                                        public void run() {
                                                makeGUI();
                                        }
                                }
                        );
                } catch (Exception exc) {
                        System.out.println("Can't create because of " + exc);
                }
        }
        private void makeGUI() {
                JTabbedPane jtp = new JTabbedPane();
                jtp.addTab("Cities", new CitiesPanel());
                jtp.addTab("Colors", new ColorsPanel());
                jtp.addTab("Flavors", new FlavorsPanel());
                add(jtp);
        }
}
```

Example: JtabbedPane p2

```
class CitiesPanel extends JPanel {
        public CitiesPanel() {
                JButton b1 = new JButton("New York");
                add(b1);
                JButton b2 = new JButton("London");
                add(b2);
                JButton b3 = new JButton("Hong Kong");
                add(b3);
                JButton b4 = new JButton("Tokyo");
                add(b4);
        }
}
class ColorsPanel extends JPanel {
        public ColorsPanel() {
                JCheckBox cb1 = new JCheckBox("Red");
                add(cb1);
                JCheckBox cb2 = new JCheckBox("Green");
                add(cb2);
                JCheckBox cb3 = new JCheckBox("Blue");
                add(cb3);
        }
}
class FlavorsPanel extends JPanel {
        public FlavorsPanel() {
                JComboBox jcb = new JComboBox();
                jcb.addItem("Vanilla");
                jcb.addItem("Chocolate");
                jcb.addItem("Strawberry");
                add(jcb);
        } }
```

JList represents a list of text items. The list of text items can be set up so that the user can choose either one item or multiple items.

JList

- supports the selection of one or more items from a list.
- A list of just about any object that can be displayed (mostly found to be string)
- JList provides several constructors:
  - JList(Object[ ] items)
- JList is based on two models:
  - ListModel
  - ListSelectionModel interface
- Most of the time you will wrap a Jlist inside a JScrollPane (makes long lists will automatically be scrollable)

JList(2)

- A JList generates a **ListSelectionEvent** when the user makes or changes or deselects a selection.
- It is handled by implementing ListSelectionListener.
  - void valueChanged(ListSelectionEvent le)
  - Here, le is a reference to the object that generated the event.
- Normally, the JList object is interrogate to determine what has occurred.
- By default, a JList allows the user to select multiple ranges of items within the list
- This behavior can be changed by calling setSelectionMode( )
  - void setSelectionMode(int mode)
  - mode (SINGLE_SELECTION, SINGLE_INTERVAL_SELECTION, and MULTIPLE_INTERVAL_SELECTION)

JList(3)

- The index of the first item selected can also be obtained by calling getSelectedIndex( ) returns the index of first selected item of the list.
  - int getSelectedIndex( )
- If no item is selected, –1 is returned.
- We can obtain the value associated with the selection by calling getSelectedValue( ):
  - Object getSelectedValue( )
- It returns a reference to the first selected value.
- If no value has been selected, it returns null.

```
// Demonstrate JList.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
/*
<applet code="JListDemo" width=200 height=120>
</applet>
*/
public class JListDemo extends JApplet {
	JList jlst;
	JLabel jlab;
	JScrollPane jscrlp;
// Create an array of cities.
	String Cities[] = { "New York", "Chicago", "Houston", "Denver", "Los Angeles", "Seattle",
			"London", "Paris", "New Delhi", "Hong Kong", "Tokyo", "Sydney" };
	public void init() {
		try {
			SwingUtilities.invokeAndWait(
				new Runnable() {
					public void run() {
						makeGUI();
					}
				}
			);
		} catch (Exception exc) {
			System.out.println("Can't create because of " + exc);
		}
	}
```

```
    private void makeGUI() {
        setLayout(new FlowLayout());
        jlst = new JList(Cities);
        // Set the list selection mode to single selection.
        jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        // Add the list to a scroll pane.
        jscrlp = new JScrollPane(jlst);
        // Set the preferred size of the scroll pane.
        jscrlp.setPreferredSize(new Dimension(120, 90));
        // Make a label that displays the selection.
        jlab = new JLabel("Choose a City");
        // Add selection listener for the list.
        jlst.addListSelectionListener(new ListSelectionListener() {
        public void valueChanged(ListSelectionEvent le) {
            // Get the index of the changed item.
            int idx = jlst.getSelectedIndex();
            // Display selection, if item was selected.
            if(idx != -1)
                jlab.setText("Current selection: " + Cities[idx]);
            else // Otherwise, reprompt.
                jlab.setText("Choose a City");
        }
    });
        // Add the list and label to the content pane.
    add(jscrlp);
    add(jlab);
}
}
```

Multiple Jpanel

```
import java.awt.*;
import javax.swing.*;
public class MultiJPanel extends JFrame {
   private JPanel mainPanel, subPanel1, subPanel2, subPanel3;
   public MultiJPanel() {
      setTitle("MultiPanel Demonstration");
      mainPanel = new JPanel(); // main panel
      mainPanel.setLayout(new GridLayout(1, 4));
      mainPanel.add(new JLabel("Main Panel", SwingConstants.CENTER));
      mainPanel.setBackground(Color.white);
      mainPanel.setBorder(BorderFactory.createLineBorder(Color.black, 1));
      subPanel1 = new JPanel();
      subPanel1.add(new JLabel("Panel One", SwingConstants.CENTER));
      subPanel1.setBackground(Color.red);
      subPanel2 = new JPanel();
      subPanel2.setBackground(Color.blue);
      subPanel2.add(new JLabel("Panel Two", SwingConstants.CENTER));
      subPanel3 = new JPanel();
      subPanel3.setBackground(Color.green);
      subPanel3.add(new JLabel("Panel Three"));
```

```
      mainPanel.add(subPanel1);
      mainPanel.add(subPanel2);
      mainPanel.add(subPanel3);
      add(mainPanel);
      setSize(400, 300);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setLocationRelativeTo(null);
      setVisible(true);
   }
   public static void main(String[] args) {
      new MultiJPanel();
   }  }
```

Self Study

- JRadioButton
- JScrollPane
- JComboBox
- JTable