Unit 5

Exploring Swing

JLabel and ImageIcon

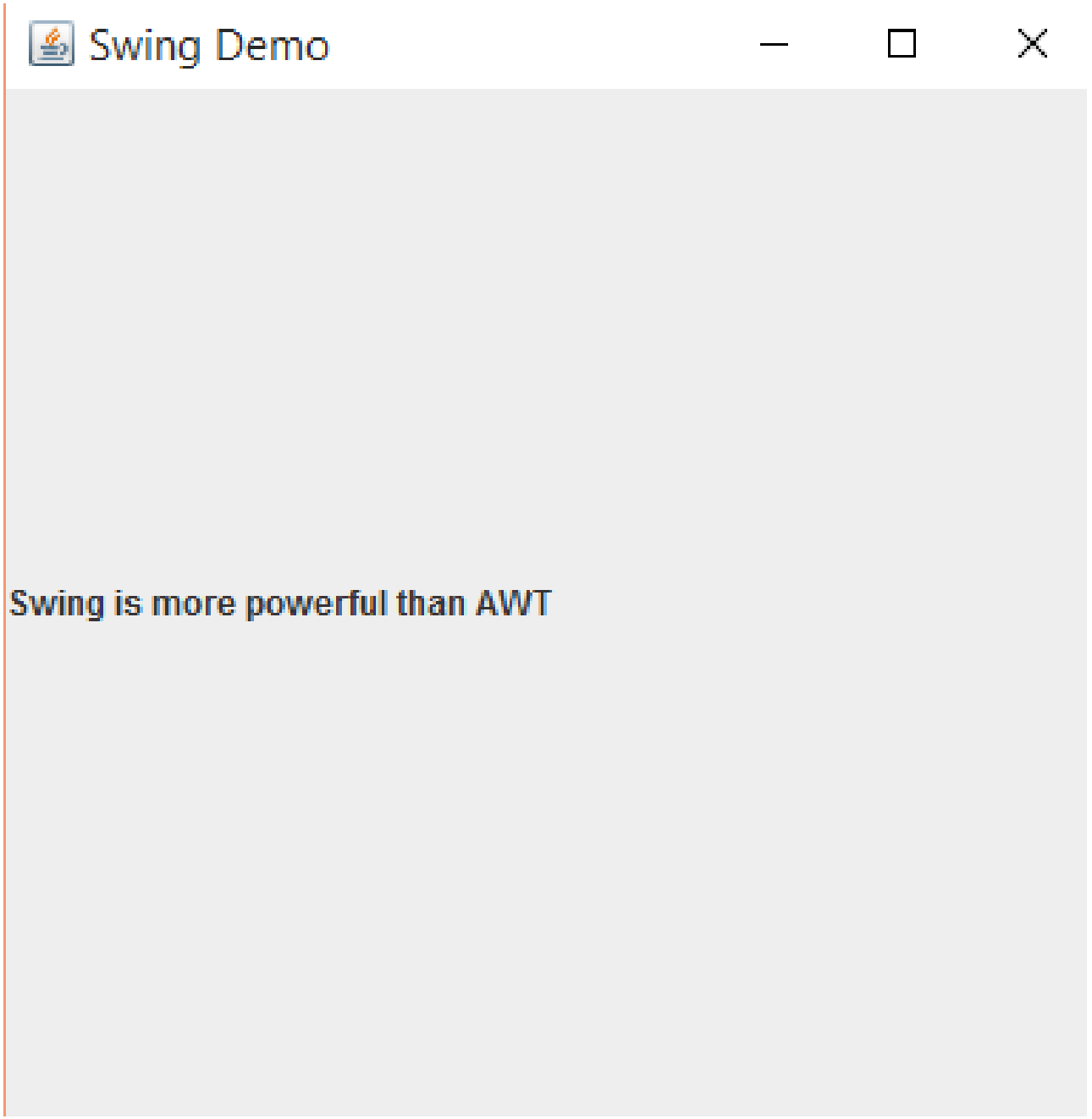
- JLabel is Swing's easiest-to-use component.
- JLabel can be used to display text and/or an icon. It is a passive component in that it does not respond to user input.
- JLabel defines several constructors. Here are three of them:
  - JLabel(Icon icon)
  - JLabel(String str)
  - JLabel(String str, Icon icon, int align)
- Here, str and icon are the text and icon used for the label.
- The align argument specifies the horizontal alignment of the text and/or icon within the dimensions of the label.
- It must be one of the following values:
- LEFT, RIGHT, CENTER, LEADING, or TRAILING.

- The easiest way to obtain an icon is to use the ImageIcon class.
- ImageIcon implements Icon and encapsulates an image.
- Thus, an object of type ImageIcon can be passed as an argument to the Icon parameter of JLabel’s constructor.
- There are several ways to provide the image, including reading it from a file or downloading it from a URL.
- Here is the ImageIcon constructor used by the example in this section:
- ImageIcon(String filename)
- It obtains the image in the file named filename.

- The icon and text associated with the label can be obtained by the following methods:
- Icon getIcon( )
- String getText( )
- The icon and text associated with a label can be set by these methods:
- void setIcon(Icon icon)
- void setText(String str)
- Here, icon and str are the icon and text, respectively. Therefore, using setText( ) it is possible to change the text inside a label during program execution.

```
import javax.swing.*;
class SwingDemo{
	public SwingDemo(){
	JFrame f=new JFrame("Swing Demo");
	f.setSize(400,400);
	f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

	JLabel l=new JLabel("Swing is more powerful than AWT");
	f.add(l);
	f.setVisible(true);
	}
	public static void main(String []args){
	new SwingDemo();
	}
}
```


Swing Demo
Swing is more powerful than AWT

```
import javax.swing.*;
class JImageIconDemo{
	public JImageIconDemo(){
	JFrame f=new JFrame("Swing Demo");
	f.setSize(400,400);
	f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
	ImageIcon image1=new ImageIcon("images.jpg");
	JLabel l=new JLabel(image1);
	f.add(l);
	f.setVisible(true);
	}
	public static void main(String []args){
	new JImageIconDemo();
	}
}
```

JTextField

- JTextField allows you to edit one line of text.
- It is derived from JTextComponent, which provides the basic functionality common to Swing text components.
- Three of JTextField's constructors are shown here:
  - JTextField(int cols)
  - JTextField(String str, int cols)
  - JTextField(String str)
- Here, str is the string to be initially presented, and cols is the number of columns in the text field. If no string is specified, the text field is initially empty. If the number of columns is not specified, the text field is sized to fit the specified string.
- To obtain the text currently in the text field, call getText( ).

The Swing Buttons

- Swing defines four types of buttons: JButton, JToggleButton, JCheckBox, and JRadioButton.
- All are subclasses of the AbstractButton class, which extends JComponent. Thus, all buttons share a set of common traits.
- AbstractButton contains many methods that allow you to control the behavior of buttons. For example, you can define different icons that are displayed for the button when it is disabled, pressed, or selected.
- Another icon can be used as a rollover icon, which is displayed when the mouse is positioned over a button. The following methods set these icons:
  - void setDisabledIcon(Icon di)
  - void setPressedIcon(Icon pi)
  - void setSelectedIcon(Icon si)
  - void setRolloverIcon(Icon ri)

- Here, di, pi, si, and ri are the icons to be used for the indicated purpose.
- The text associated with a button can be read and written via the following methods:
- String getText( )
- void setText(String str)
- Here, str is the text to be associated with the button.
- A button generates an action event when it is pressed. Other events are possible.

JButton

- The JButton class provides the functionality of a push button. You have already seen a simple form of it in the preceding chapter.
- JButton allows an icon, a string, or both to be associated with the push button. Three of its constructors are shown here:
- JButton(Icon icon)
- JButton(String str)
- JButton(String str, Icon icon)
- Here, str and icon are the string and icon used for the button.
- When the button is pressed, an ActionEvent is generated.
- Using the ActionEvent object passed to the actionPerformed( ) method of the registered ActionListener, you can obtain the action command

- You can obtain the action command by calling getActionCommand( ) on the event object. It is declared like this:
- String getActionCommand( )
- The action command identifies the button. Thus, when using two or more buttons within the same application, the action command gives you an easy way to determine which button was pressed.

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.
- In addition to creating standard toggle buttons, JToggleButton is a superclass for two other Swing components that also represent two-state controls.
- These are JCheckBox and JRadioButton, which are described later in this chapter.
- Thus, JToggleButton defines the basic functionality of all

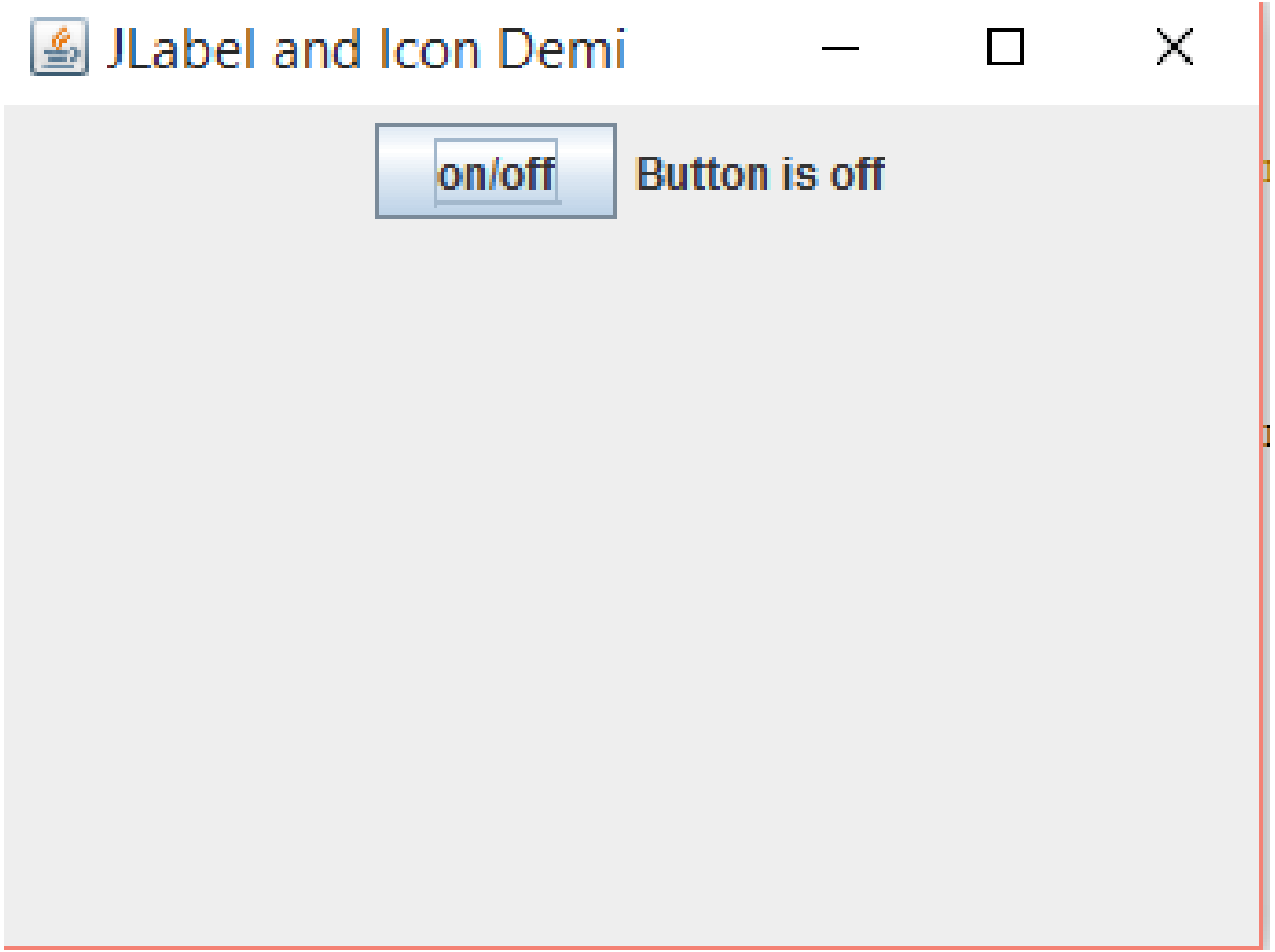
- The one used by the example in this section is shown here:
- JToggleButton(String str)
- This creates a toggle button that contains the text passed in str. By default, the button is in the off position. Other constructors enable you to create toggle buttons that contain images, or images and text.
- Like JButton, JToggleButton generates an action event each time it is pressed. Unlike JButton, however, JToggleButton also generates an item event. This event is used by those components that support the concept of selection. 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.

- The easiest way to determine a toggle button’s state is by calling the isSelected( ) method (inherited from AbstractButton) on the button that generated the event. It is shown here:
- boolean isSelected( )
- It returns true if the button is selected and false otherwise.
- Here is an example that uses a toggle button. Notice how the item listener works. It simply calls isSelected( ) to determine the button’s state.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class JToggleButtonDemo {
    public JToggleButtonDemo(){
        JFrame frame = new JFrame(“JToggleButton Demo");
        frame.setSize(200, 200);

        frame.setLayout(new FlowLayout());
        JLabel l1=new JLabel("Button is off");
        JToggleButton jtbtn=new JToggleButton("on/off");
         jtbtn.addItemListener(new ItemListener() {
```

```
@Override
            public void itemStateChanged(ItemEvent e) {
                if(jtbtn.isSelected()){
                    l1.setText("button is on");
                }else{
                    l1.setText("button is off");
                }
            }
        });
frame.add(jtbtn);
        frame.add(l1);
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        new JToggleButtonDemo();
    }
}
```


JLabel and Icon Demi
on/off
Button is off

Check Boxes

- The JCheckBox class provides the functionality of a check box. Its immediate superclass is JToggleButton, which provides support for two-state buttons, as just described.
- JCheckBox defines several constructors. The one used here is
- JCheckBox(String str)
- It creates a check box that has the text specified by str as a label. Other constructors let you specify the initial selection state of the button and specify an icon.
- When the user selects or deselects a check box, an ItemEvent is generated. You can obtain a reference to the JCheckBox that generated the event by calling getItem( ) on the ItemEvent passed to the itemStateChanged( ) method defined by ItemListener. The easiest way to determine the selected state of a

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

public class JCheckBoxDemo2 implements ItemListener {

    JCheckBox c1, c2;
    JLabel status;

    public JCheckBoxDemo2() {
        JFrame frame = new JFrame("Checkbox Demo");
        frame.setLayout(new GridLayout(2, 1));
        frame.setSize(600, 400);
```

```
JPanel p = new JPanel(new FlowLayout());
JPanel p1 = new JPanel(new FlowLayout());
c1 = new JCheckBox("singing");
c2 = new JCheckBox("dancing");

JLabel hobbies = new JLabel("Hobbies");
status = new JLabel();
c1.addItemListener(this);
c2.addItemListener(this);

p.add(hobbies);
p.add(c1);
p.add(c2);
p1.add(status);
```

```
        frame.add(p);
        frame.add(p1);

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
;
        frame.setVisible(true);
    }
 public void itemStateChanged(ItemEvent ie) {
        String singing = "", dancing = "";
        if (c1.isSelected()) {
            singing = c1.getText();
        }
        if (c2.isSelected()) {
            dancing = c2.getText();
```

```
        status.setText("Your Hobbies :::" + singing + " " +
dancing);
    }
     public static void main(String[] args) {
        new JCheckBoxDemo2();
    }
}
```

Radio Buttons

- Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any one time. They are supported by the JRadioButton class, which extends JToggleButton. JRadioButton provides several constructors.
- JRadioButton(String str)
- Here, str is the label for the button. Other constructors let you specify the initial selection state of the button and specify an icon.
- In order for their mutually exclusive nature to be activated, radio buttons must be configured into a group.
- Only one of the buttons in the group can be selected at any time. For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected.
- A button group is created by the ButtonGroup class. Its default constructor is invoked for this purpose. Elements

- A JRadioButton generates action events, item events, and change events each time the button selection changes.
- Most often, it is the action event that is handled, which means that you will normally implement the ActionListener interface.
- Recall that the only method defined by ActionListener is actionPerformed( ). Inside this method, you can use a number of different ways to determine which button was selected.
- First, you can check its action command by calling getActionCommand( ). By default, the action command is the same as the button label, but you can set the action command to something else by calling setActionCommand( ) on the radio button.
- Second, you can call getSource( ) on the ActionEvent object and check that reference against the buttons.
- Third, you can check each radio button to find out which one is currently selected by calling isSelected(

Example

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

public class RadioButtonDemo implements
ActionListener {
    private JLabel l1;
    public RadioButtonDemo(){
        JFrame frame = new JFrame(“JRadioButton Demo”);
        frame.setLayout(new FlowLayout());
        frame.setSize(200, 200);
        JRadioButton jr1=new JRadioButton("Male");
        JRadioButton jr2=new JRadioButton("Female");
```

```
        l1=new JLabel();
        ButtonGroup bg=new ButtonGroup();
        bg.add(jr1);
        bg.add(jr2);
        jr1.addActionListener(this);
        jr2.addActionListener(this);
        frame.add(jr1);
        frame.add(jr2);
        frame.add(l1);

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CL
OSE);
        frame.setVisible(true);
```

```
   public void actionPerformed(ActionEvent ae){
        l1.setText("You selected
"+ae.getActionCommand());
    }
    public static void main(String[] args) {
        new RadioButtonDemo();
    }
}
```

JTabbedPane

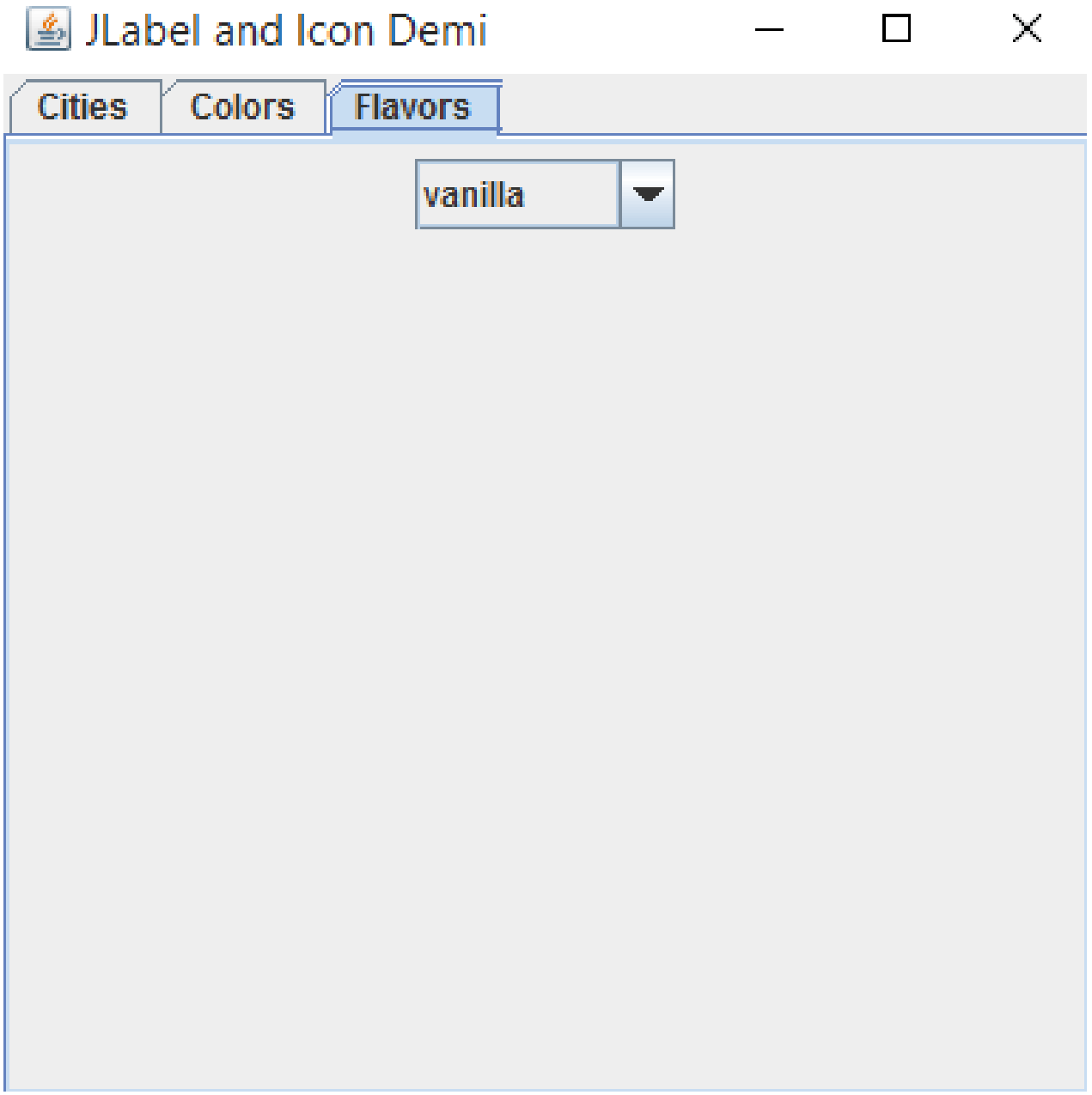
- JTabbedPane encapsulates a tabbed pane. It manages a set of components by linking them with tabs.
- Selecting a tab causes the component associated with that tab to come to the forefront.
- JTabbedPane defines three constructors.
- We will use its default constructor, which creates an empty control with the tabs positioned across the top of the pane.
- The other two constructors let you specify the location of the tabs, which can be along any of the four sides.
- JTabbedPane uses the SingleSelectionModel model. Tabs are added by calling addTab( ). Here is one of its forms:
- 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, the component added to a tab is a JPanel that contains a

- The general procedure to use a tabbed pane is outlined here:
  - 1. Create an instance of JTabbedPane.
  - 2. Add each tab by calling addTab( ).
  - 3. Add the tabbed pane to the content pane.

```
import javax.swing.*;
public class JTabbedPaneDemo {
    public JTabbedPaneDemo() {
        JFrame frame = new JFrame();
        frame.setSize(400, 400);
        JTabbedPane jtp = new JTabbedPane();
        jtp.addTab("Cities", new CitiesPanel());
        jtp.addTab("Colors", new ColorPanel());
        jtp.addTab("Flavors", new FlavorPanel());
        frame.add(jtp);
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        new JTabbedPaneDemo();
    }
}
```

```
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("HongKong");
        add(b3);
        JButton b4=new JButton("Tokyo");
        add(b4);
    }
}
class ColorPanel extends JPanel{
    public ColorPanel(){
```

```
        JCheckBox cb1=new JCheckBox("Red");
         add(cb1);
         JCheckBox cb2=new JCheckBox("Green");
         add(cb2);
         JCheckBox cb3=new JCheckBox("Blue");
         add(cb3);
    }
}
class FlavorPanel extends JPanel{
    public FlavorPanel(){
         JComboBox jcb=new JComboBox();
         jcb.addItem("vanilla");
         jcb.addItem("Chocolate");
         jcb.addItem("Strawberry");
         add(jcb);
    } }
```


JLabel and Icon Demi
Cities
Colors
Flavors
vanilla

JScrollPane

- JScrollPane is a lightweight container that automatically handles the scrolling of another component.
- The component being scrolled can be either an individual component, such as a table, or a group of components contained within another lightweight container, such as a JPanel.
- In either case, if the object being scrolled is larger than the viewable area, horizontal and/or vertical scroll bars are automatically provided, and the component can be scrolled through the pane.
- Because JScrollPane automates scrolling, it usually eliminates the need to manage individual scroll bars.
- The viewable area of a scroll pane is called the viewport.
- It is a window in which the component being scrolled is displayed.
- Thus, the viewport displays the visible portion of the

- In its default behavior, a JScrollPane will dynamically add or remove a scroll bar as needed.
- For example, if the component is taller than the viewport, a vertical scroll bar is added. If the component will completely fit within the viewport, the scroll bars are removed.
- JScrollPane defines several constructors. The one used in this chapter is shown here:
  - JScrollPane(Component comp)
- The component to be scrolled is specified by comp. Scroll bars are automatically displayed when the content of the pane exceeds the dimensions of the viewport.
- Here are the steps to follow to use a scroll pane:
- 1. Create the component to be scrolled.
- 2. Create an instance of JScrollPane, passing to it the object to scroll.
- 3. Add the scroll pane to the content pane.

```
import java.awt.*
import javax.swing.*;
public class JScrollPaneDemo {
    public JScrollPaneDemo() {
        JFrame frame = new JFrame("");
       // frame.setLayout(new GridLayout(2, 1));
        frame.setSize(400,400);
        JPanel p = new JPanel(new GridLayout(20,20));
        int b=0;
           for(int i=0;i<20;i++){
               for(int j=0;j<20;j++){
                   p.add(new JButton("Button"+b));
                   ++b;
               }
           }
```

```
JScrollPane jsp=new JScrollPane(p);
        frame.add(jsp);

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
;
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        new JScrollPaneDemo();
    }
}
```

JList

- In Swing, the basic list class is called JList.
- It supports the selection of one or more items from a list.
- However, beginning with JDK 7, JList was made generic and is now declared like this:
  - class JList<E>
- Here, E represents the type of the items in the list.
- JList provides several constructors. The one used here is
  - JList(E[ ] items)
- This creates a JList that contains the items in the array specified by items.

- Although a JList will work properly by itself, most of the time you will wrap a JList inside a JScrollPane.
- This way, long lists will automatically be scrollable, which simplifies GUI design.
- It also makes it easy to change the number of entries in a list without having to change the size of the JList component.
- A JList generates a ListSelectionEvent when the user makes or changes a selection.
- This event is also generated when the user deselects an item. It is handled by implementing ListSelectionListener.
- This listener specifies only one method, called valueChanged( ), which is shown here:
- void valueChanged(ListSelectionEvent le)

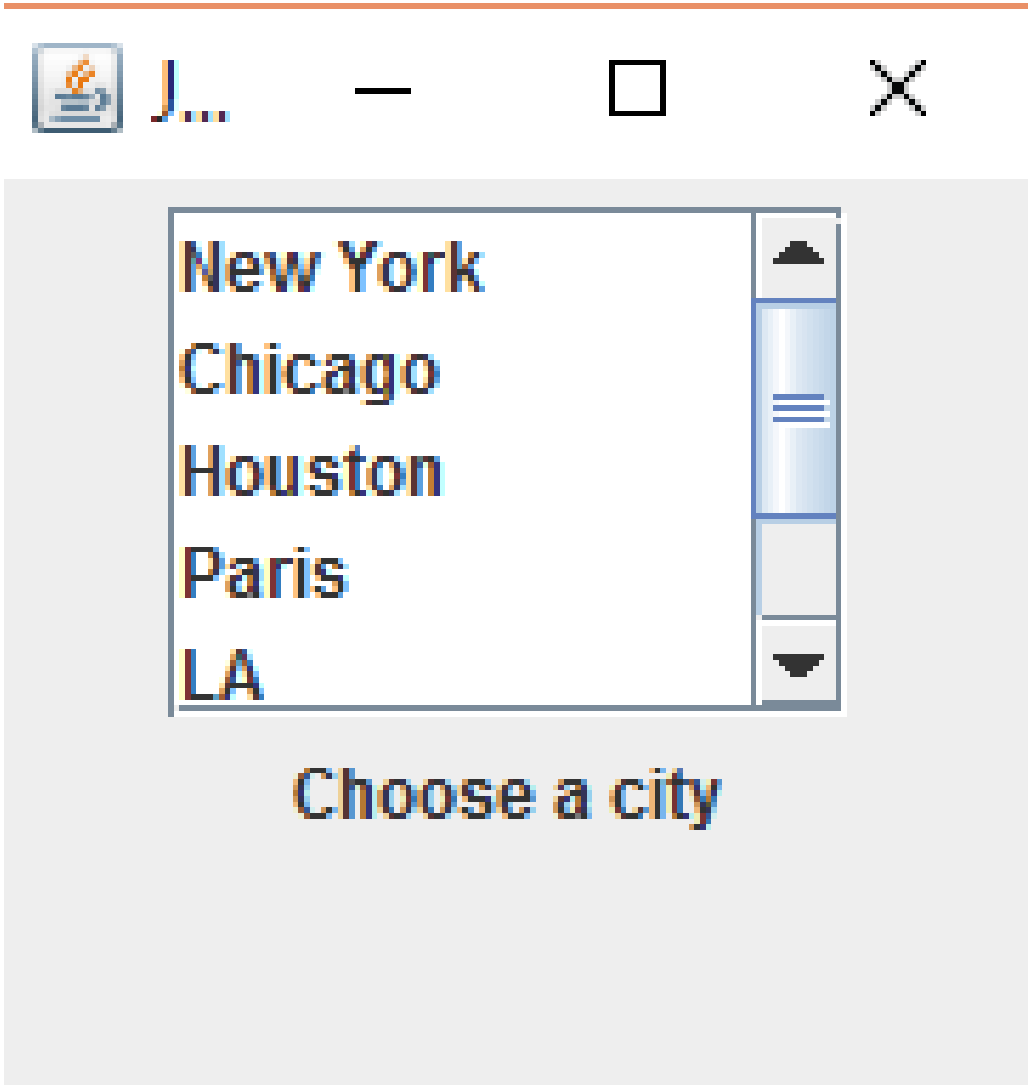
- Both ListSelectionEvent and ListSelectionListener are packaged in javax.swing.event.
- By default, a JList allows the user to select multiple ranges of items within the list, but you can change this behavior by calling setSelectionMode( ), which is defined by JList. It is shown here:
- void setSelectionMode(int mode)
- Here, mode specifies the selection mode. It must be one of these values defined by ListSelectionModel: SINGLE_SELECTION SINGLE_INTERVAL_SELECTION MULTIPLE_INTERVAL_SELECTION
- The default, multiple-interval selection, lets the user select multiple ranges of items within a list.
- With single-interval selection, the user can select one range of items.
- With single selection, the user can select only a single

- Of course, a single item can be selected in the other two modes, too. It’s just that they also allow a range to be selected.
- You can obtain the index of the first item selected, which will also be the index of the only selected item when using single-selection mode, by calling getSelectedIndex( ), shown here:
- int getSelectedIndex( )
- Indexing begins at zero. So, if the first item is selected, this method will return 0.
- If no item is selected, –1 is returned. Instead of obtaining the index of a selection, you can obtain the value associated with the selection by calling getSelectedValue( ):
- E getSelectedValue( )
- It returns a reference to the first selected value. If no

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

public class JListDemo {
    JList<String> jlist;
    JLabel city;
    JScrollPane jspane;
    public JListDemo() {
        JFrame frame = new JFrame();
        frame.setSize(200, 200);
        frame.setLayout(new FlowLayout());
        String cities[] = {"New York", "Chicago", "Houston", "Paris",
"LA", "kathmandu", "New Delhi"};
        jlist=new JList<>(cities);
        jlist.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        jspane=new JScrollPane(jlist);
        jspane.setPreferredSize(new Dimension(120,90));
        city=new JLabel("Choose a city");
```

```
jlist.addListSelectionListener(new ListSelectionListener() {
public void valueChanged(ListSelectionEvent e) {
                int idx=jlist.getSelectedIndex();
                if(idx!=-1){
                    city.setText("Current Selection:"+cities[idx]);
                }else{
                    city.setText("Choose a city");
                }
            }
        });
        frame.add(jspane);
        frame.add(city);
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        new JListDemo();
    }
}
```


J...
New York
Chicago
Houston
Paris
LA
Choose a city

JComboBox

- Swing provides a combo box (a combination of a text field and a drop-down list) through the JComboBox class.
- A combo box normally displays one entry, but it will also display a drop-down list that allows a user to select a different entry.
- You can also create a combo box that lets the user enter a selection into the text field.
- In the past, the items in a JComboBox were represented as Object references. However, beginning with JDK 7, JComboBox was made generic and is now declared like this: class JComboBox<E>
- Here, E represents the type of the items in the combo box. The JComboBox constructor used by the example is shown here:
- JComboBox(E[ ] items)
- Here, items is an array that initializes the combo box.

- In addition to passing an array of items to be displayed in the drop-down list, items can be dynamically added to the list of choices via the addItem( ) method, shown here:
- void addItem(E obj)
- Here, obj is the object to be added to the combo box. This method must be used only with mutable combo boxes.
- JComboBox generates an action event when the user selects an item from the list.
- JComboBox also generates an item event when the state of selection changes, which occurs when an item is selected or deselected.
- Thus, changing a selection means that two item events will occur: one for the deselected item and another for the selected item.
- Often, it is sufficient to simply listen for action events, but both event types are available for your use.
- One way to obtain the item selected in the list is to call getSelectedItem( ) on the combo box. It is shown here:
- Object getSelectedItem( )
- You will need to cast the returned value into the type of object stored in the list

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

public class JComboBoxDemo extends JFrame{
    JLabel jlab;
    JComboBox<String> jcb;
    String
timepiece[]={"morning","Afternoon","evening","night"};
    public JComboBoxDemo(){
        setLayout(new FlowLayout());
        jcb=new JComboBox<String>(timepiece);
        add(jcb);
        JLabel jlab=new JLabel("Choose time");
        add(jlab);
        jcb.addActionListener(new ActionListener() {
```

```
@Override
        public void actionPerformed(ActionEvent e) {
            String s=(String)jcb.getSelectedItem();
            jlab.setText("Your selection: "+s);
        }
    });
    setSize(200,200);
    setVisible(true);
  }
  public static void main(String[] args) {
    new JComboBoxDemo();
  }
}
```

JTable

- JTable is a component that displays rows and columns of data.
- You can drag the cursor on column boundaries to resize columns.
- You can also drag a column to a new position.
- Jtable offers substantial functionality that is easy to use—especially if you simply want to use the table to present data in a tabular format.
- JTable has many classes and interfaces associated with it. These are packaged in javax.swing.table.
- At its core, JTable is conceptually simple. It is a component that consists of one or more columns of information.
- At the top of each column is a heading. In addition to describing the data in a column, the heading also provides the mechanism by which the user can change the size of a column or change the location of a column within the table.
- JTable does not provide any scrolling capabilities of its own. Instead, you will normally wrap a JTable inside a JScrollPane.

- JTable supplies several constructors. The one used here is
- JTable(Object data[ ][ ], Object colHeads[ ])
- Here, data is a two-dimensional array of the information to be presented, and colHeads is a one-dimensional array with the column headings. JTable relies on three models. The first is the table model, which is defined by the TableModel interface.
- This model defines those things related to displaying data in a two dimensional format.
- The second is the table column model, which is represented by TableColumnModel.
- JTable is defined in terms of columns, and it is TableColumnModel that specifies the characteristics of a column.
- These two models are packaged in javax.swing.table.
- The third model determines how items are selected, and it is specified by the ListSelectionModel, which was described when JList was discussed.

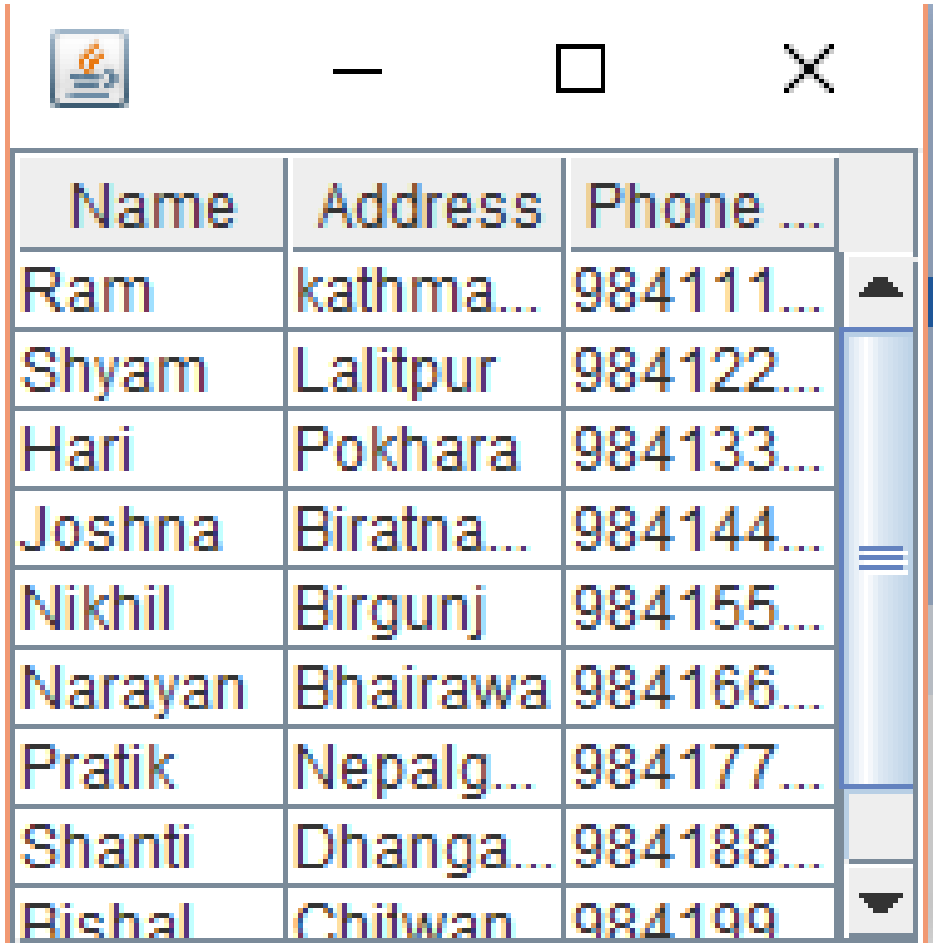
- A JTable can generate several different events.
- The two most fundamental to a table’s operation are ListSelectionEvent and TableModelEvent.
- A ListSelectionEvent is generated when the user selects something in the table.
- By default, JTable allows you to select one or more complete rows, but you can change this behavior to allow one or more columns, or one or more individual cells to be selected.
- A TableModelEvent is fired when that table’s data changes in some way. Handling these events requires a bit more work than it does to handle the events generated by the previously described components and is beyond the scope of this book.
- However, if you simply want to use JTable to display data (as the following example does), then you don’t need to handle any events.
- Here are the steps required to set up a simple JTable that can be used to display data:
- 1. Create an instance of JTable.
- 2. Create a JScrollPane object, specifying the table as the object to scroll.
- 3. Add the table to the scroll pane.
- 4. Add the scroll pane to the content pane.

```
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
public class JTableDemo extends JFrame{

    public JTableDemo() {
        String[] colhead = {"Name", "Address", "Phone number"};
        Object[][] data = {{"Ram", "kathmandu", "9841111111"},
        {"Shyam", "Lalitpur", "9841222222"},
        {"Hari", "Pokhara", "984133333"},
        {"Joshna", "Biratnagar", "9841444444"},
        {"Nikhil", "Birgunj", "9841555555"},
        {"Narayan", "Bhairawa", "9841666666"},
        {"Pratik", "Nepalgunj", "9841777777"},
        {"Shanti", "Dhangadi", "9841888888"},
        {"Bishal", "Chitwan", "9841999999"},
        {"MAdhav", "Argakhanchi", "9841234567"}};
```

```
JTable table=new JTable(data,colhead);
        JScrollPane jsp=new JScrollPane(table);
        add(jsp);
        setSize(200,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);

    }
    public static void main(String[] args) {
        new JTableDemo();
    }
}
```


Name
Address
Phone ...
Ram
kathma...
984111...
Shyam
Lalitpur
984122...
Hari
Pokhara
984133...
Joshna
Biratna...
984144...
Nikhil
Birgunj
984155...
Narayan
Bhairawa
984166...
Pratik
Nepalg...
984177...
Shanti
Dhanga...
984188...
Bishal
Chitwan
984199

SWING MENUS

- JMenuBar
- JMenu
- JMenuItem
  - JRadioButtonMenuItem
  - JCheckBoxMenuItem
  - Jseperator
  - JPopupMenu

Example

```
import java.awt.event.*;
import javax.swing.*;
public class MenuDesign extends JFrame {
    JMenuBar mb;
    JMenu file, edit, view;
    JMenuItem makenew, open, save, saveas, exit;
    JSeparator js;
    JCheckBoxMenuItem cr1, cr2;
    JRadioButtonMenuItem jr1, jr2;

    public MenuDesign() {
        mb = new JMenuBar();
        file = new JMenu("File");
        file.setMnemonic('F');
```

```
        edit = new JMenu("Edit");
        view = new JMenu("View");
        makenew = new JMenuItem("New", 'N');
        KeyStroke ctrlNKeyStroke =
KeyStroke.getKeyStroke("control N");
        makenew.setAccelerator(ctrlNKeyStroke);

        open = new JMenuItem("Open",'O');
        save = new JMenuItem("Save");
        saveas = new JMenuItem("SaveAs");
        exit = new JMenuItem("Exit");
        js = new JSeparator();
        cr1 = new JCheckBoxMenuItem("Copy");
        cr2 = new JCheckBoxMenuItem("Cut");
        jr1 = new JRadioButtonMenuItem("View Full
Screen");
        jr2 = new JRadioButtonMenuItem("View Half
```

```
ButtonGroup bg = new ButtonGroup();
bg.add(jr1);
bg.add(jr2);

file.add(makenew);
file.add(open);
file.add(save);
file.add(saveas);
file.add(js);
file.add(exit);

edit.add(cr1);
edit.add(cr2);

view.add(jr1);
view.add(jr2);
```

```
    mb.add(file);
    mb.add(edit);
    mb.add(view);
    setJMenuBar(mb);
    makenew.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(null, "Hello");
        }});
    setSize(400, 400);
    setLocationRelativeTo(null);
    setDefaultCloseOperation(3);
    setVisible(true);
}
```

```
public static void main(String[] args) {
        try
{          UIManager.setLookAndFeel("com.jtattoo.plaf.acryl.Acry
lLookAndFeel");
        } catch (Exception e) {
        }
        new MenuDesign();
    }


}
```

Example 2

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

public class JMenuDemo extends JFrame {

    public JMenuDemo() {
        JMenuBar menubar = new JMenuBar();
        JMenu file = new JMenu("File");
        JMenu Edit = new JMenu("Edit");
        file.setMnemonic(KeyEvent.VK_F);
        JMenuItem item1 = new JMenuItem("New");
        item1.setMnemonic(KeyEvent.VK_N);
        JMenuItem item2 = new JMenuItem("Open");
        JMenuItem item3 = new JMenuItem("Save");
```

```
  JMenuItem item4 = new JMenuItem("SaveAs");
  JMenuItem item5 = new JMenuItem("Exit");
 JCheckBoxMenuItem item6 = new
JCheckBoxMenuItem("Save all");
 JRadioButtonMenuItem item7 = new
JRadioButtonMenuItem("Print");
  item5.setMnemonic(KeyEvent.VK_X);
  item5.setDisplayedMnemonicIndex(1);
   item5.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                //System.exit(0);
            JOptionPane.showMessageDialog(null, "You have
click on exit");
```

```
file.add(item1);
file.add(item2);
file.add(item3);
file.add(item4, 1);
file.addSeparator();
file.add(item5);
file.add(item6);
file.add(item7);
menubar.add(file);
menubar.add(Edit);
setJMenuBar(menubar);
item4.setEnabled(false);
item3.setEnabled(false);
```

```
        JPopupMenu jp = new JPopupMenu();
        JMenuItem copy = new JMenuItem("Copy");
        JMenuItem cut = new JMenuItem("cut");
        JMenuItem paste = new JMenuItem("paste");
        jp.add(copy);
        jp.add(cut);
        jp.add(paste);
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent me) {
                if (me.isPopupTrigger()) {
                    jp.show(me.getComponent(), me.getX(),
me.getY());
                }
            }
```

```
public void mouseReleased(MouseEvent me) {
                if (me.isPopupTrigger()) {
                    jp.show(me.getComponent(), me.getX(),
me.getY());
                }
            }
        });
}
public static void main(String[] args) {
        try {
//
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeel
ClassName());
```

```
UIManager.setLookAndFeel("com.jtattoo.plaf.aluminium.AluminiumLookAndFeel");
        } catch (Exception e) {
            e.printStackTrace();
        }
        JMenuDemo a = new JMenuDemo();
        a.setResizable(false);
        a.setSize(800, 600);
        a.setLocationRelativeTo(null);
        a.setDefaultCloseOperation(3);
        a.setVisible(true);
    }
}
```

JPopUpMenu Example

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

public class PopUpMenuDesign extends JFrame {
    public PopUpMenuDesign(){
        JPopupMenu jp=new JPopupMenu();
        JMenuItem copy=new JMenuItem("Copy");
        JMenuItem cut=new JMenuItem("Cut");
        JMenuItem paste=new JMenuItem("Paste");

        jp.add(copy);
        jp.add(cut);
        jp.add(paste);
```

```
addMouseListener(new MouseAdapter() {
        public void mouseReleased(MouseEvent me){
            if(me.isPopupTrigger()){
                jp.show(me.getComponent(), me.getX(),
me.getY());
            }

        }
    });
    setSize(400, 400);
    setDefaultCloseOperation(3);
    setVisible(true);
  }
  public static void main(String[] args) {
    new PopUpMenuDesign();
  }
}
```

Java JOptionPane

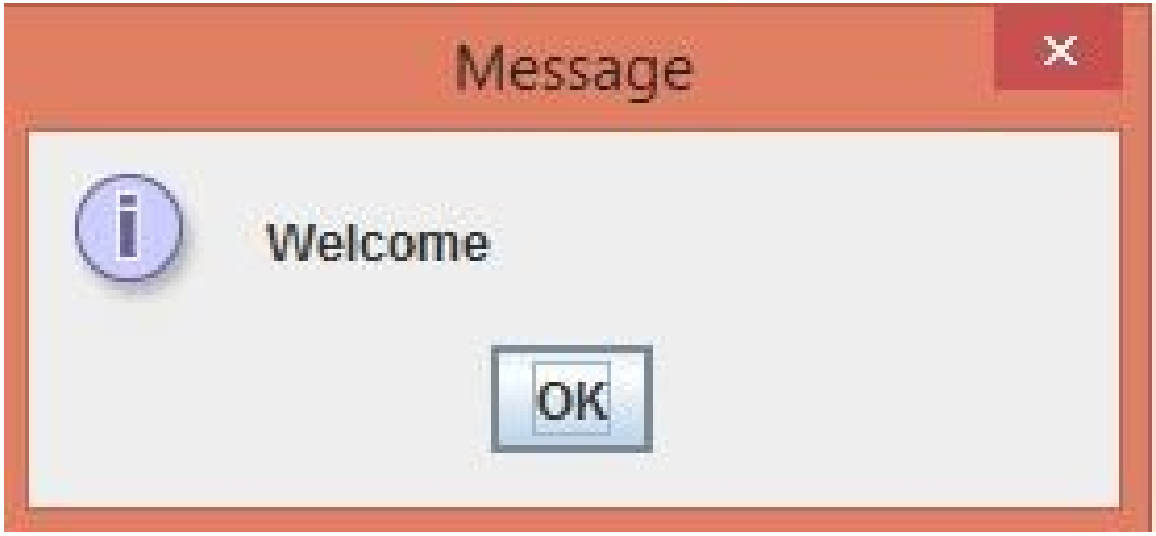
- The JOptionPane class is used to provide standard dialog boxes such as message dialog box, confirm dialog box and input dialog box.
- The showMessageDialog method is used to display a message dialog.
- An input dialog is a quick and simple way to ask the user to enter data
- A confirm dialog box typically asks the user a Yes or No question. By default a Yes button, a No button, and a Cancel button are displayed.
- These dialog boxes are used to display information or get input from the user.

Methods	Description
JDialog createDialog(String title)	It is used to create and return a new parentless JDialog with the specified title.
static void showMessageDialog(Component parentComponent, Object message)	It is used to create an information-message dialog titled "Message".
static void showMessageDialog(Component parentComponent, Object message, String title, int messageType)	It is used to create a message dialog with given title and messageType.
static int showConfirmDialog(Component parentComponent, Object message)	It is used to create a dialog with the options Yes, No and Cancel; with the title, Select an Option.
static String showInputDialog(Component parentComponent, Object message)	It is used to show a question-message dialog requesting input from the user parented to parentComponent.
void setInputValue(Object newValue)	It is used to set the input value that was selected or input by the user.

Example

```
import javax.swing.JOptionPane;
public class JOptionPaneDemo {


    public JOptionPaneDemo() {
      JOptionPane.showMessageDialog(null,
"Welcome");
    }
    public static void main(String[] args) {
        new JOptionPaneDemo();
    }
}
```


Message
Welcome
OK

Types

- JOptionPane.showMessageDialog(Component parentComponent, String message, String title, int messageType, Icon icon);

Brings up an information message dialog.

- JOptionPane.showConfirmDialog(Component parentComponent, String message, String title, int optionType, int messageType, Icon icon);

Brings up a dialog with the options Yes, No and Cancel and also returns the integer value

- JOptionPane.showInputDialog(Component parentComponent, String message, String title, int messageType);

Shows a question-message dialog requesting input

- JOptionPane.showInternalConfirmDialog(Component parentComponent, String message, String title, int optionType, int messageType, Icon icon);

Brings up an internal dialog panel with the options Yes, No and Cancel and also returns the value according to the choice of the user.

- JOptionPane.showInternalInputDialog(Component parentComponent, String message, String title, int messageType);

Shows an internal question-message dialog requesting input from the user parented to parentComponent.

- JOptionPane.showInternalMessageDialog(Componen t parentComponent, String message, String title, int messageType, Icon icon);

Brings up an internal confirmation dialog panel. The dialog is a information-message dialog.

- JOptionPane.showOptionDialog(Component parentComponent, String message, String Title, optionType, messageType, Icon icon, Array[] options, Object initialValue);

Brings up a dialog with a specified icon, where the initial choice is determined by the initialValue parameter and the number of choices is determined by the optionType parameter. In options, we can pass an array of objects indicating the possible choices the user can make.

Eg: JOptionPane.showOptionDialog(f, "Hello", "Test",JOptionPane.DEFAULT_OPTION, JOptionPane.PLAIN_MESSAGE, null, options, null);

- JAPPLET

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JAppletDemo.class" width=200
height=200>
</applet>
*/
public class JAppletDemo extends JApplet{
    public void init(){
        setLayout(new FlowLayout());
        JButton b1=new JButton("Alpha");
        JButton b2=new JButton("Beta");
        JLabel label=new JLabel("Press a button");
```

```
        b1.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
              label.setText("You have pressed Alpha");
           }
        });
        b2.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
              label.setText("You have pressed Beta");
           }
        });
        add(b1);
        add(b2);
        add(label);
    }
}
```