Module V
Java Swing

Dr. Zahid Ansari

Java Swing

- A Set of classes that provides powerful and flexible GUI components
- Lightweight- Not built on native window-system windows.
- Much bigger set of built-in controls. Trees, image buttons, tabbed panes, sliders, toolbars, etc.
- More customizable. Can change border, text alignment, or add image to almost any control. Can customize how minor features are drawn.
- "Pluggable" look and feel. Can change look and feel at runtime, or design own look and feel.
- Many miscellaneous new features.

The Origin of Swing

- It is a response to deficiencies present in Java’s original GUI system AWT.
- AWT translates various components into their corresponding platform specific equivalents. Since AWT components use native code resources they are referred to as heavyweights
  - Because of variations between operating systems a component might look, or even act differently on different platforms.
  - Look and feel of each component is fixed (defined by the underlying platform) and can not be changed easily.
  - Use of heavy weight components caused some frustrating restrictions
    - Heavyweight component is always rectangular and opaque

Swing is built on the AWT

- Swing is built on the AWT
  - It eliminates a number of limitations inherent in AWT
  - It does not replace AWT

Two Key Swing Features

- Swing components are lightweight
  - They are written entirely in Java and they do not map directly to platform specific peers
  - Light weight components are more efficient and more flexible
  - Look and feel is determined by swing and not by the underlying OS

- Swing supports a pluggable look and feel
  - It is possible to separate the look and feel of component from the logic of the component
  - It is possible to “plug in” a new look and feel for any given component without creating any side effects in the code that uses that component.
  - The look and feel can be changed dynamically at run time

Components and Containers

- Swing GUI consists of two kinds of items
  - Components
    - A component is an independent visual control, such as push button, lable or list.
  - Containers
    - A container is a special kind of component that holds other components
    - In order for a component to be displayed it must be held within a container
- All SWING GUIs have at least one container
    - Since containers are components , a container can also hold other containers
    - This enables SWINGS to define containment hierarchy, at the top of which there is top level component

Components

- In general Swing components are derived from JComponent class
  - JComponent supports the pluggable look and feel

- Swing components are represented by classes defined within the package

  javax.swing

- Example of Swing component classes

  JApplet, JButton, JCheckBox, JColorChooser, JComboBox, JComponent, JDialog, JFrame, JLable, JList, JMenu, JPanel, JRadioButton, JScrollBar, JTable, JTree, JWindow etc

Containers

- Swing defines two types of containers
  - Top level containers
    - Example: JFrame, JApplet, JWindow, JDialog
    - They do not inherit from JComponent, instead they inherit from AWTs Component and Container classes
    - They are heavyweight containers
    - They appear at the top of the containment hierarchy
    - Every containment hierarchy must begin with a top level component
      - The most commonly used for the applications is JFrame
      - The one used for applets is JApplet
  - Light Weight containers
    - Inherit fro JComponent
    - Example JPanel
    - Often used to organize and manage group of related components
    - Light weight containers can be contained within another container

Swing Packages

- Swing is a very large subsystem and consists of many 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.palf.multi
Javax.swing.plaf.synth	Javax.swing.table	Javax.swing.text
Javax.swing.text.html	Javax.swing.text.html.parse	Javax.swing.text.rtf
Javax.swing.tree	Javax.swing.undo	

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);
    }
```

A simple Swing application.

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

Handle an event in a Swing program.

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

class EventDemo {
      JLabel jlab;
      EventDemo() {
            // Create a new JFrame container.
            JFrame jfrm = new JFrame("An Event Example");

            // Specify FlowLayout for the layout manager.
            jfrm.setLayout(new FlowLayout());

            // Give the frame an initial size.
            jfrm.setSize(220, 90);

            // Terminate the program when the user closes the application.
            jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

            // Make two buttons.
            JButton jbtnAlpha = new JButton("Alpha");
            JButton jbtnBeta = new JButton("Beta");
```

Handle an event in a Swing program.

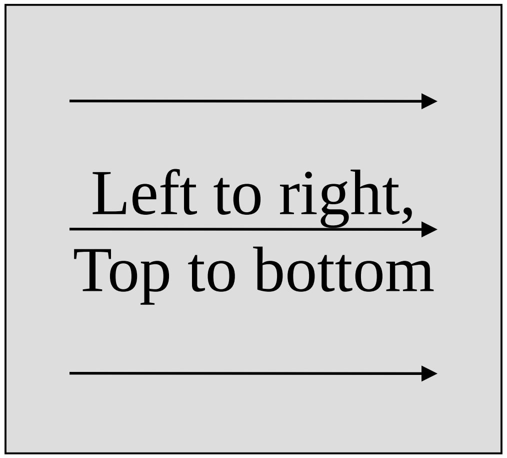
```
        // Add action listener for Alpha and Beta
         jbtnAlpha.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent ae) {
                        jlab.setText("Alpha was pressed.");
                }
        });
        jbtnBeta.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent ae) {
                        jlab.setText("Beta was pressed.");
                }
        });
        // Add the buttons to the content pane.
        jfrm.add(jbtnAlpha);
        jfrm.add(jbtnBeta);

        // Create a text-based label.
        jlab = new JLabel("Press a button.");
        // Add the label to the content pane.
        jfrm.add(jlab);

        // Display the frame.
        jfrm.setVisible(true);
    }
```

Handle an event in a Swing program

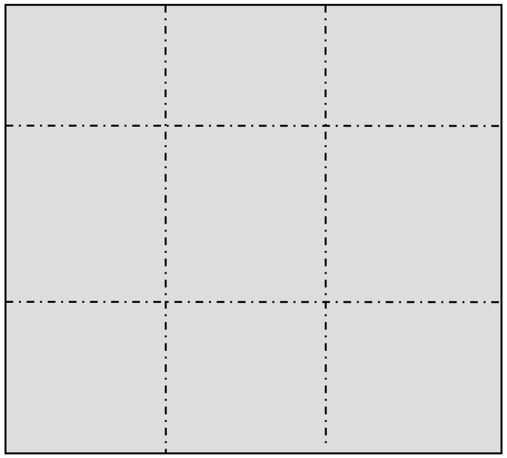
```
    public static void main(String args[]) {
    // Create the frame on the event dispatching thread.
    SwingUtilities.invokeLater(new Runnable() {
                    public void run() {
                              new EventDemo();
                    }
    });
    }
}
```

Layout Heuristics

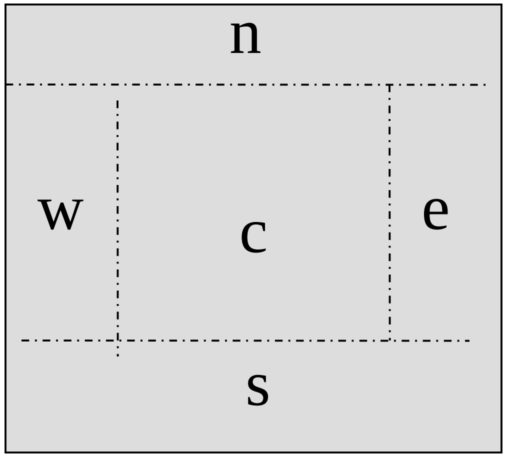
FlowLayout



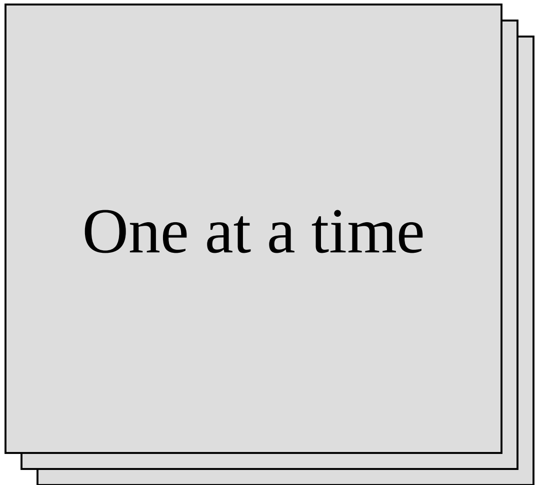
GridLayout



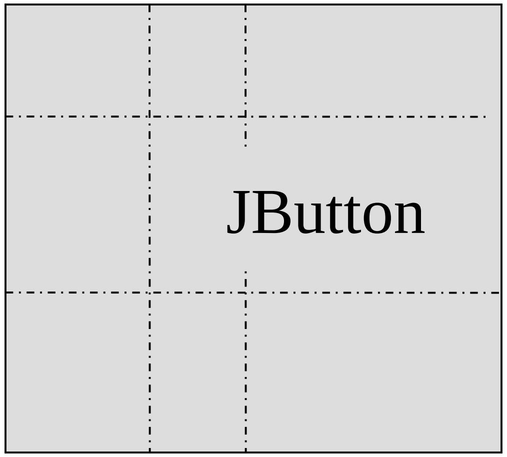
BorderLayout



CardLayout



GridBagLayout



A simple Swing-based applet

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

public class MySwingApplet extends JApplet {
    JButton jbtnAlpha;
    JButton jbtnBeta;
    JLabel jlab;

    // Initialize the applet.
    public void init() {
        try {
            SwingUtilities.invokeAndWait( new Runnable () {
                public void run() {
                    makeGUI(); // initialize the GUI
                }
            });
        } catch(Exception exc) {
                System.out.println("Can't create because of "+ exc);
        }
    }
```

A simple Swing-based applet

```
private void makeGUI() {
// Set the applet to use flow layout.
setLayout(new FlowLayout());
// Make two buttons.
jbtnAlpha = new JButton("Alpha");
jbtnBeta = new JButton("Beta");

// Add action listener for Alpha and Beta
 jbtnAlpha.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent le) {
                    jlab.setText("Alpha was pressed.");
          }
});
// Add action listener for Beta.
 jbtnBeta.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent le) {
                    jlab.setText("Beta was pressed.");
          }
});
// Add the buttons to the content pane.
add(jbtnAlpha);
add(jbtnBeta);

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

Paint lines to a panel.

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

// This class extends JPanel. It overrides the paintComponent() method so
// that random lines are plotted in the panel.
class PaintPanel extends JPanel {
    Insets ins;                 // holds the panel's insets
    Random rand;                // used to generate random numbers

    // Construct a panel.
    PaintPanel() {
        // Put a border around the panel.
        setBorder(BorderFactory.createLineBorder(Color.RED, 5));
        rand = new Random();
    }
```

Paint lines to a panel.

```
// Override the paintComponent() method.
protected void paintComponent(Graphics g) {
          // Always call the superclass method first.
           super.paintComponent(g);
           int x, y, x2, y2;
          // Get the height and width of the component.
          int height = getHeight();
          int width = getWidth();
          // Get the insets.
          ins = getInsets();

          // Draw ten lines whose endpoints are randomly generated.
          for(int i=0; i < 10; i++) {
                    // Obtain random coordinates that define the endpoints of each line
                    x = rand.nextInt(width-ins.left);
                    y = rand.nextInt(height-ins.bottom);
                    x2 = rand.nextInt(width-ins.left);
                    y2 = rand.nextInt(height-ins.bottom);
                    // Draw the line.
                    g.drawLine(x, y, x2, y2);
          }
}
```

Paint lines to a panel

```
// Demonstrate painting directly onto a panel.
class PaintDemo {
    PaintPanel pp;

    PaintDemo() {
            // Create a new JFrame container.
            JFrame jfrm = new JFrame("Paint Demo");
            // Give the frame an initial size.
            jfrm.setSize(200, 150);
            // Terminate the program when the user closes the application.
            jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

            // Create the panel that will be painted.
            pp = new PaintPanel();
            // Add the panel to the content pane.  Because the default  border layout is
            // used, the panel will automatically be  sized to fit the center region.
            jfrm.add(pp);

            // Display the frame.
            jfrm.setVisible(true);
 }
```

Paint lines to a panel

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

JLable and ImageIcon

- JLable defines
  - several constructors
    - JLable(Icon *icon*)
    - JLable(String *str*)
    - JLable(String *str*, Icon *icon*, int *align*)
  - Methods
    - Icon getIcon()
    - String getText()
    - void setIcon(Icon *icon*)
    - void setText(String *str*)
- ImageIcon constructor
    - ImageIcon(String *filename*)

JLabel and ImageIcon.

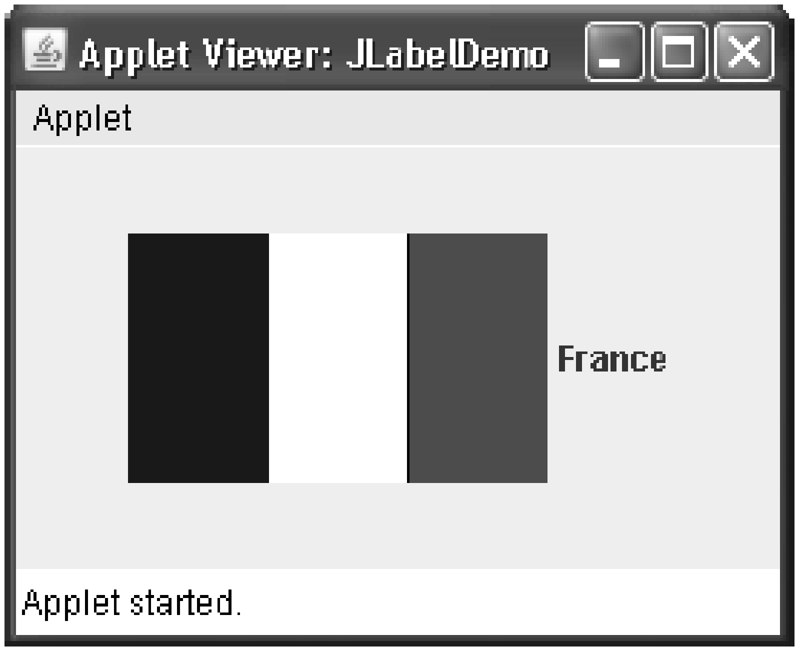
```
import java.awt.*;
import javax.swing.*;

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);
    }
}
```

JLabel and ImageIcon.

```
private void makeGUI() {

        // Create an icon.
        ImageIcon ii = new ImageIcon("france.gif");

        // Create a label.
        JLabel jl = new JLabel("France", ii, JLabel.CENTER);

        // Add the label to the content pane.
        add(jl);
    }
}
```

JTextField

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

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);
     }
}
```

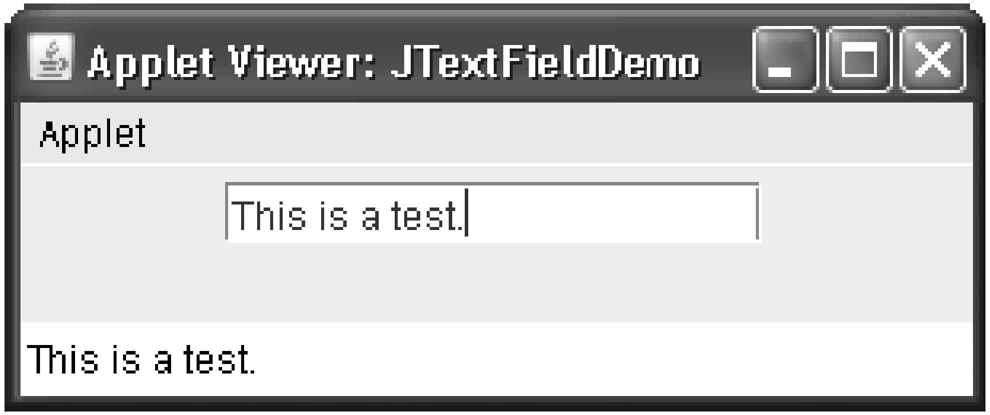

Applet Viewer: JLabelDemo
Applet
France
Applet started.

JText Field

- Simplest and most widely used Swing text component
  - Constructors
    - JTextField(int *cols*)
    - JTextField(String *str*, int *cols*)
    - JTextField(String *str*)
  - Methods
    - String getText()

JTextField

```
    private void makeGUI() {

        // Change to flow layout.
        setLayout(new FlowLayout());

        // Add text field to content pane.
        jtf = new JTextField(15);
        add(jtf);
        jtf.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent ae) {
                        // Show text when user presses ENTER.
                         showStatus(jtf.getText());
                }
        });
    }
}
```



```
This is a test.
```

```
This is a test.
```

Swing Buttons

- Swing defines four types of buttons
  - JButton
  - JToggleButton
  - JCheckBox
  - JRadioButton

  All are subclasses of the **AbstractButton** class
- The text associated with a button can be read and written via the following methods:

  String getText( )

  void setText(String *str*)

JButton

- The **JButton** class provides the functionality of a push button.
- **JButton** allows an icon, a string, or both to be associated with the push button.
- Three of its constructors are:

  JButton(Icon *icon*)

  JButton(String *str*)

  JButton(String *str*, Icon *icon*)

- When the button is pressed, an **ActionEvent** is generated.
- This **ActionEvent** object passed to the **actionPerformed( )** method of the registered **ActionListener**
- The following example displays four push buttons and a label. Each button displays an icon that represents the flag of a country. When a button is pressed, the name of that country is displayed in the label.

Icon-based JButton

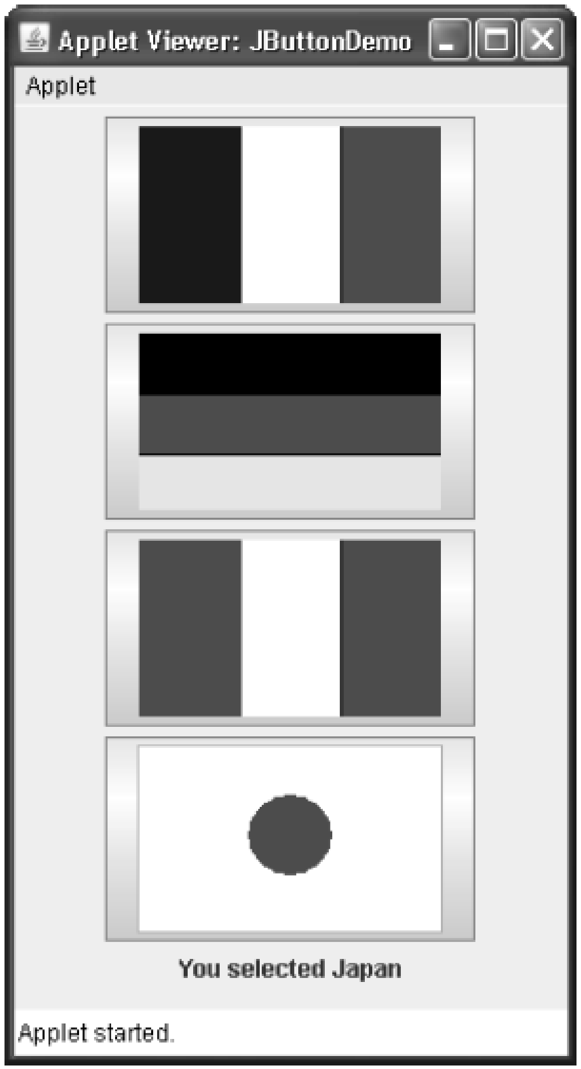
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

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);
      }
}
```

Icon-based JButton

```
private void makeGUI() {
        // Change to flow layout.
        setLayout(new FlowLayout());

        // Add buttons to content pane.
        ImageIcon france = new ImageIcon("france.gif");
        JButton jb = new JButton(france);
        jb.setActionCommand("France");
        jb.addActionListener(this);
        add(jb);

        ImageIcon china = new ImageIcon("china.gif");
        jb = new JButton(china);
        jb.setActionCommand("China");
        jb.addActionListener(this);
        add(jb);

        ImageIcon italy = new ImageIcon("italy.gif");
        jb = new JButton(italy);
        jb.setActionCommand("Italy");
        jb.addActionListener(this);
        add(jb);
```

Icon-based JButton

```
            ImageIcon japan = new ImageIcon("japan.gif");
            jb = new JButton(japan);
            jb.setActionCommand("Japan");
            jb.addActionListener(this);
            add(jb);

            // Create and add the label to content pane.
            jlab = new JLabel("Choose a Country Map");
            add(jlab);
      }

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


Applet Viewer: JButtonDemo
Applet
You selected Japan
Applet started.

JToggleButton

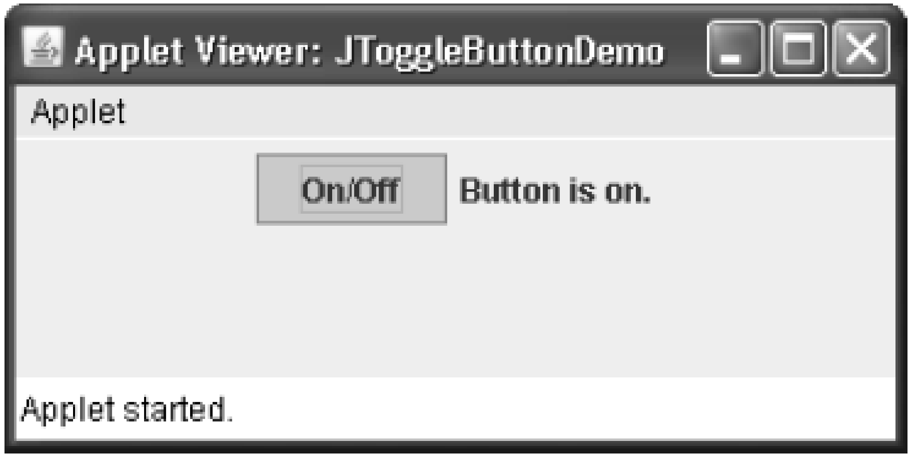
- A Toggle button has two states
  - Pushed
  - Released
- When you press the toggle button it stays pressed, when you press a second time it releases
- By default the button is in the off position
- JToggleButton is super class of JCheckBox and JRadioButton classes

JToggleButton

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

public class JToggleButtonDemo extends JApplet {
    JLabel jlab;
    JToggleButton jtbn;
    public void init() {
      try {
                SwingUtilities.invokeAndWait( new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                } );
      } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
      }
}
```

JToggleButton

```
    private void makeGUI() {
            // Change to flow layout.
            setLayout(new FlowLayout());

            // Create a label.
            jlab = new JLabel("Button is off.");
            // Make a toggle button.
            jtbn =  new JToggleButton("On/Off");

            // Add an item listener for the toggle button.
            jtbn.addItemListener(new ItemListener() {
                    public void itemStateChanged(ItemEvent ie) {
                            if(jtbn.isSelected())
                                    jlab.setText("Button is on.");
                            else
                                    jlab.setText("Button is off.");
                            }
            });
            // Add the toggle button and label to the content pane.
            add(jtbn);
            add(jlab);
    }
}
```



Button is on.

Applet started.

JCheckBox

- 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. One 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. **ItemEvent is** passed to the **itemStateChanged( )** method defined by **ItemListener**.

JCheckBox Example

- The following example displays four check boxes and a label. When the user clicks a check box, an **ItemEvent** is generated.
- Inside the **itemStateChanged( )** method, **getItem( )** is called to obtain a reference to the **JCheckBox** object that generated the event.
- Next, a call to **isSelected( )** determines if the box was selected or cleared.
- The **getText( )** method gets the text for that check box and uses it to set the text inside the label.

JCheckbox

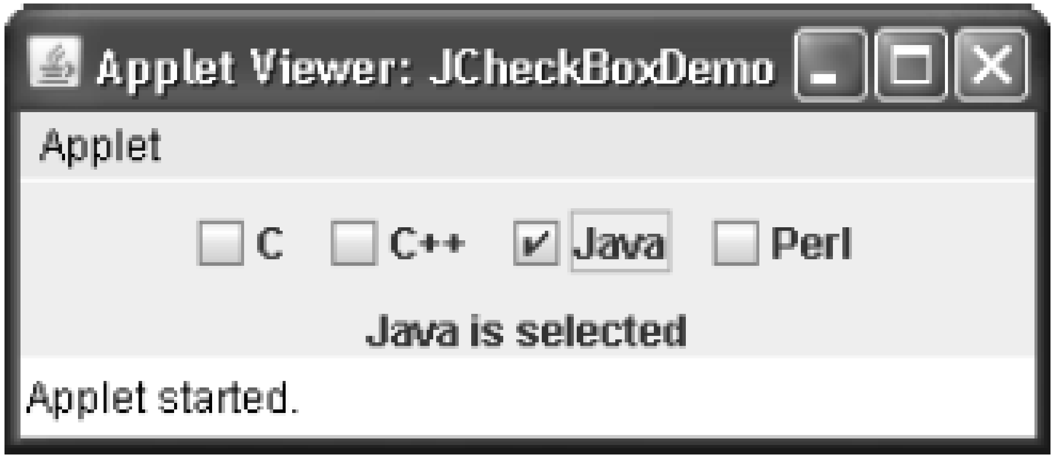
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

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);
      }
}
```

JCheckbox

```
private void makeGUI() {
            setLayout(new FlowLayout());

           // Add check boxes to the content pane.
           JCheckBox cb = new JCheckBox("C");
           cb.addItemListener(this);
           add(cb);

           cb = new JCheckBox("C++");
           cb.addItemListener(this);
           add(cb);

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

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

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

JCheckbox.

```
  // 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");
   }
}
```



```
Java is selected
Applet started.
```

JRadioButton

- Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any one time.
- **JRadioButton** class extends **JToggleButton**.
- **JRadioButton** provides several constructors. The one is shown here:

  JRadioButton(String *str*)
- A button group is created by the **ButtonGroup** class. Elements are then added to the button group using method:

  void add(AbstractButton *ab*)
- A**JRadioButton** generates action events, item events, and change events each time the button selection changes.
- In following example three radio buttons are created and then added to a button group. Pressing a radio button generates an action event, which is handled by **actionPerformed()**.
- Within that handler, the **getActionCommand( )** method gets the text that is associated with the radio button and uses it to set the text within a label.

JRadioButton

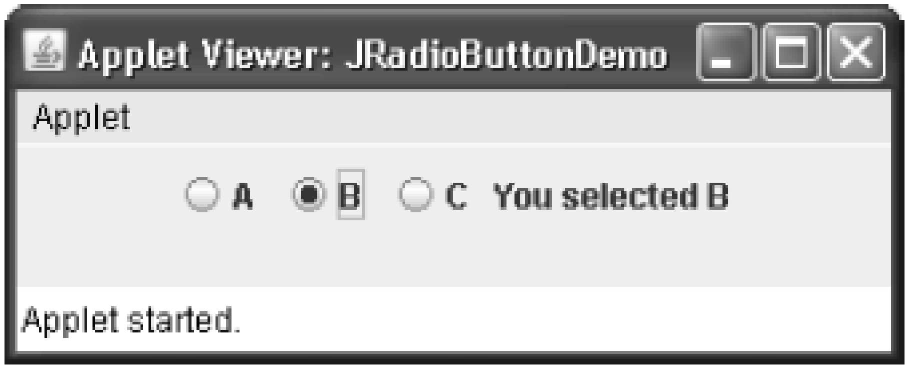
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JRadioButtonDemo 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);
     }
}
```

JRadioButton

```
private void makeGUI() {
      // Change to flow layout.
     setLayout(new FlowLayout());

     // Create radio buttons and add them to content pane.
     JRadioButton b1 = new JRadioButton("A");
     b1.addActionListener(this);
     add(b1);
     JRadioButton b2 = new JRadioButton("B");
     b2.addActionListener(this);
     add(b2);
     JRadioButton b3 = new JRadioButton("C");
     b3.addActionListener(this);
     add(b3);

     // Define a button group.
     ButtonGroup bg = new ButtonGroup();
     bg.add(b1);  bg.add(b2); bg.add(b3);

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

JRadioButton

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



You selected B

Applet started.

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** default constructor creates an empty control with the tabs positioned across the top of the pane.
- Tabs are added by calling **addTab( )**. Here is one of its forms:

  void addTab(String *name*, Component *comp*)
- 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.

JTabbedPane Example

The following example illustrates a tabbed pane. The first tab is titled “Cities” and contains four buttons. Each button displays the name of a city. The second tab is titled “Colors” and contains three check boxes. Each check box displays the name of a color. The third tab is titled “Flavors” and contains one combo box. This enables the user to select one of three flavors.

JTabbedPane

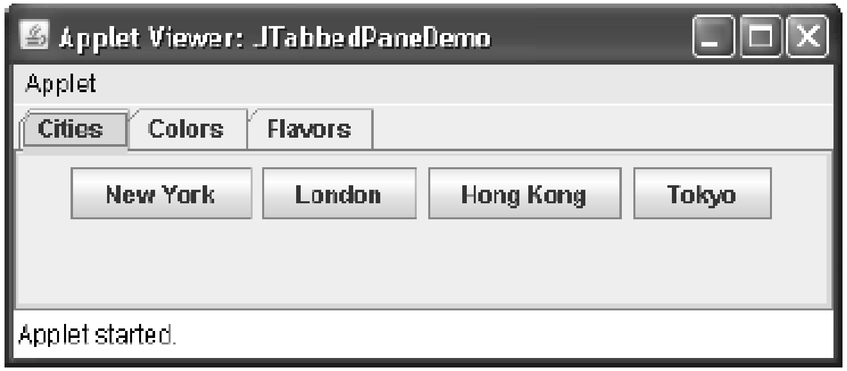
```
import javax.swing.*;

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);
     }
}
```

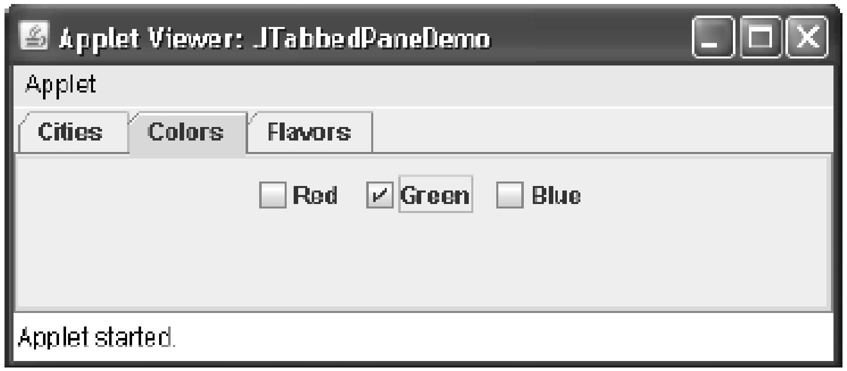
JTabbedPane

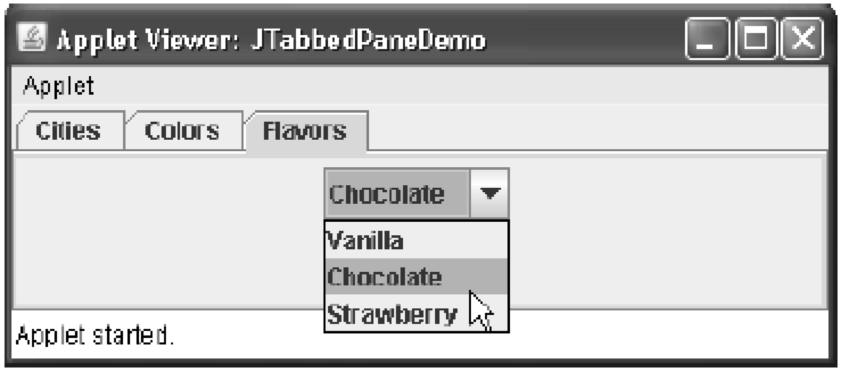
```
    private void makeGUI() {
            JTabbedPane jtp = new JTabbedPane();
            jtp.addTab("Cities", new CitiesPanel());
            jtp.addTab("Colors", new ColorsPanel());
            jtp.addTab("Flavors", new FlavorsPanel());
            add(jtp);
    }
}

// Make the panels that will be added to the tabbed pane.
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);
    }
}
```

JTabbedPane

```
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);
      }
}
```


Applet Viewer: JTabbedPaneDemo
Applet
Cities
Colors
Flavors
New York
London
Hong Kong
Tokyo
Applet started.


Applet Viewer: JTabbedPaneDemo
Applet
Cities
Colors
Flavors
Red
Green
Blue
Applet started.


Applet Viewer: JTabbedPaneDemo
Applet
Cities
Colors
Flavors
Chocolate
Vanilla
Chocolate
Strawberry
Applet started.

JScrollPane

- **JScrollPane** is a lightweight container that automatically handles the scrolling of another component.
- The component being scrolled can either be an individual component, such as a table, or a group of components contained within another lightweight container, such as a **JPanel**.
- 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.
- 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 using

     JScrollPane(Component *comp*)
  3. Add the scroll pane to the content pane.

JScrollPane Example

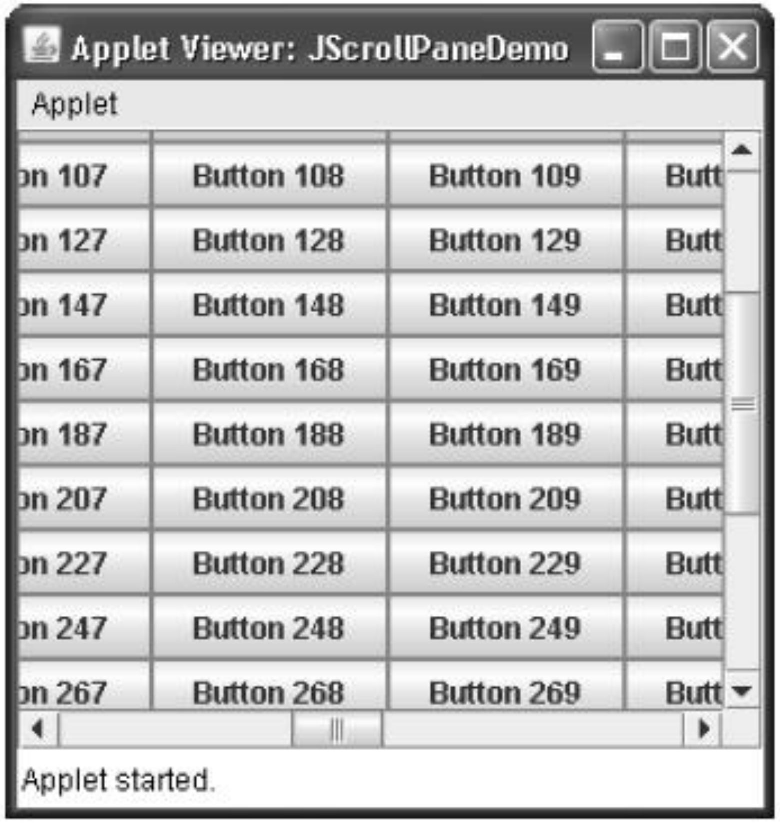
- The following example illustrates a scroll pane. First, a **JPanel** object is created, and 400 buttons are added to it, arranged into 20 columns. This panel is then added to a scroll pane, and the scroll pane is added to the content pane. Because the panel is larger than the viewport, vertical and horizontal scroll bars appear automatically. You can use the scroll bars to scroll the buttons into view.

JScrollPane

```
import java.awt.*;
import javax.swing.*;
public class JScrollPaneDemo 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);
    }
}
```

JScrollPane

```
    private void makeGUI() {
    `       // Add 400 buttons to a panel.
            JPanel jp = new JPanel();
            jp.setLayout(new GridLayout(20, 20));
            int b = 0;
            for(int i = 0; i < 20; i++) {
                    for(int j = 0; j < 20; j++) {
                            jp.add(new JButton("Button " + b));
                            ++b;
                    }
            }
            // Create the scroll pane.
            JScrollPane jsp = new JScrollPane(jp);

            // Add the scroll pane to the content pane. Because the default border layout
            // is  used, the scroll pane will be added to the center.
            add(jsp, BorderLayout.CENTER);
    }
}
```



Applet started.

JList

- In Swing, the basic list class is called **JList**. It supports the selection of one or more items from a list. Although the list often consists of strings, it is possible to create a list of any object that can be displayed.
- **JList** provides several constructors. Following constructor creates a **JList** that contains the items in the array specified by *items*.

  JList(Object[ ] *items*)
- A **JList** generates a **ListSelectionEvent** when the user makes or changes a selection or deselects an item. It is handled by implementing

  ListSelectionListener. This listener specifies only one method, called **valueChanged( )**
- 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( )**
- Following applet demonstrates a simple **JList**, which holds a list of cities. Each time a city is selected in the list, a **ListSelectionEvent** is generated, which is handled by the **valueChanged( )** method defined by **ListSelectionListener**. It responds by obtaining the index of the selected item and displaying the name of the selected city in a label.

JList

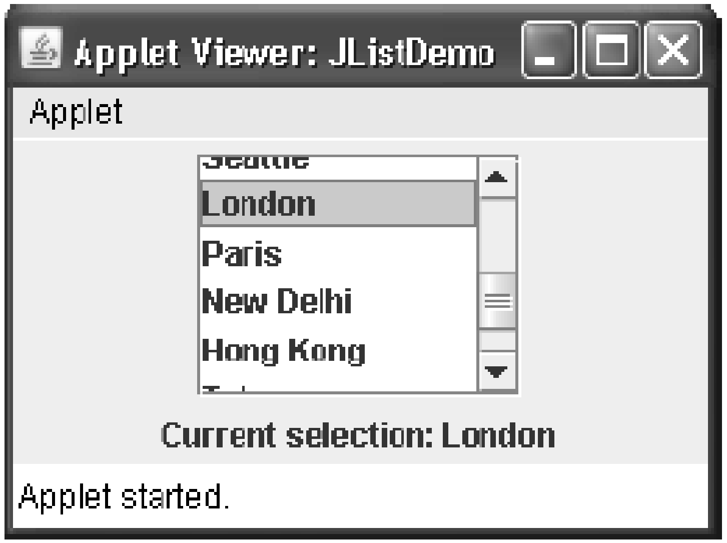
```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
public class JListDemo extends JApplet {
      JList jlst;
      JLabel jlab;
      JScrollPane jscrlp;
      // Create an array of cities.
      String Cities[] = {  "New York", "Chicago", "Houston",
                "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);
        }
      }
```

JList

```
private void makeGUI() {

    // Change to flow layout.
    setLayout(new FlowLayout());

    // Create a JList.
    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");
```

JList

```
    // 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 // Othewise, reprompt.
                                    jlab.setText("Choose a City");

        }
    });
        // Add the list and label to the content pane.
    add(jscrlp);
    add(jlab);
    }
}
```



Current selection: London

Applet started.

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.
- The following **JComboBox** constructor initializes the combo box using *items* array

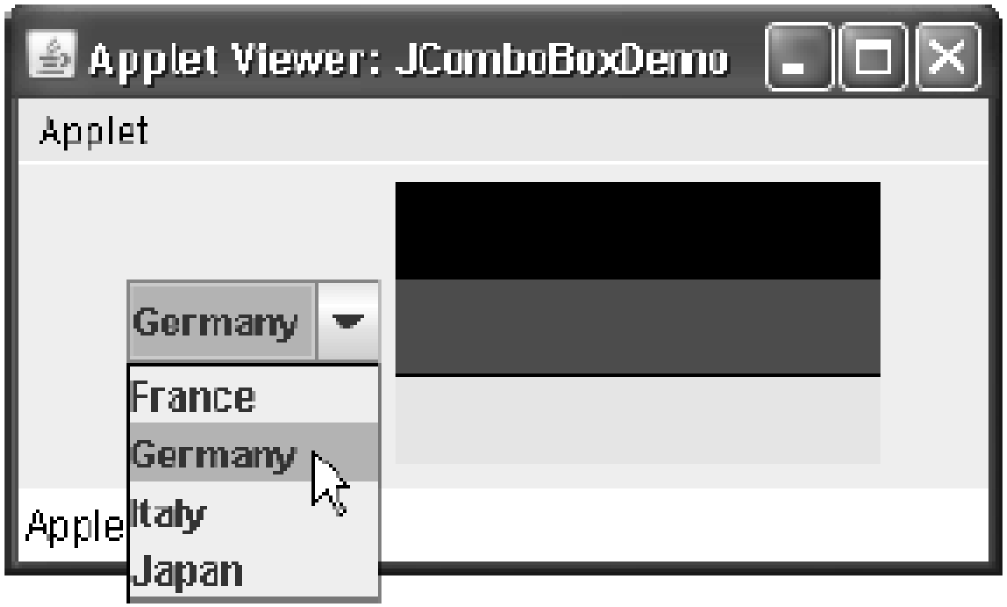
  JComboBox(Object[ ] *items*)
- **JComboBox** generates an action event when the user selects an item from the list. **JComboBox** also generates an item event when the state of selection changes, which occurs when an item is selected or deselected.
- One way to obtain the item selected in the list is to call **getSelectedItem( )** on the combobox.
- The following example demonstrates the combo box. The combo box contains entries for “France,” “Germany,” “Italy,” and “Japan.” When a country is selected, an icon-based label is updated to display the flag for that country.

JComboBox

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class JComboBoxDemo extends JApplet {
    JLabel jlab;
    ImageIcon france, germany, italy, japan;
    JComboBox jcb;
    String flags[] = { "France", "Germany", "Italy", "Japan" };

    public void init() {
      try {
            SwingUtilities.invokeAndWait( new Runnable() {
                        public void run() {
                              makeGUI();
                        }
                  } );
      } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
    }
}
```

JComboBox

```
    private void makeGUI() {
        // Change to flow layout.
        setLayout(new FlowLayout());

        // Instantiate a combo box and add it to the content pane.
         jcb = new JComboBox(flags);
        add(jcb);

         // Handle selections.
        jcb.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent ae) {
                        String s = (String) jcb.getSelectedItem();
                        jlab.setIcon(new ImageIcon(s + ".gif"));
                }
        });

        // Create a label and add it to the content pane.
        jlab = new JLabel(new ImageIcon("france.gif"));
        add(jlab);
    }
}
```


Applet Viewer: JComboBoxDemo
Applet
Germany
France
Germany
Italy
Japan
Apple

JTree

- A *tree* is a component that presents a hierarchical view of data. The user has the ability to expand or collapse individual subtrees in this display.
- Trees are implemented in Swing by the **JTree** class. One of the Jtree constructor is shown below

  JTree(TreeNode *tn*) // *tn* is root node

- **JTree** does not provide any scrolling capabilities of its own. Instead, a **JTree** is typically placed within a **JScrollPane**. This way, a large tree can be scrolled through a smaller viewport.

Here are the steps to follow to use a tree:

1. Create an instance of **JTree**.

2. Create a **JScrollPane** and specify the tree as the object to be scrolled.

3. Add the tree to the scroll pane.

4. Add the scroll pane to the content pane.

JTree Example

- The following example illustrates how to create a tree and handle selections. The program creates a **DefaultMutableTreeNode** instance labeled “Options.” This is the top node of the tree hierarchy. Additional tree nodes are then created, and the **add( )** method is called to connect these nodes to the tree. A reference to the top node in the tree is provided as the argument to the **JTree** constructor. The tree is then provided as the argument to the **JScrollPane** constructor. This scroll pane is then added to the content pane. Next, a label is created and added to the content pane. The tree selection is displayed in this label. To receive selection events from the tree, a **TreeSelectionListener** is registered for the tree. Inside the **valueChanged( )** method, the path to the current selection is obtained and displayed.

JTree

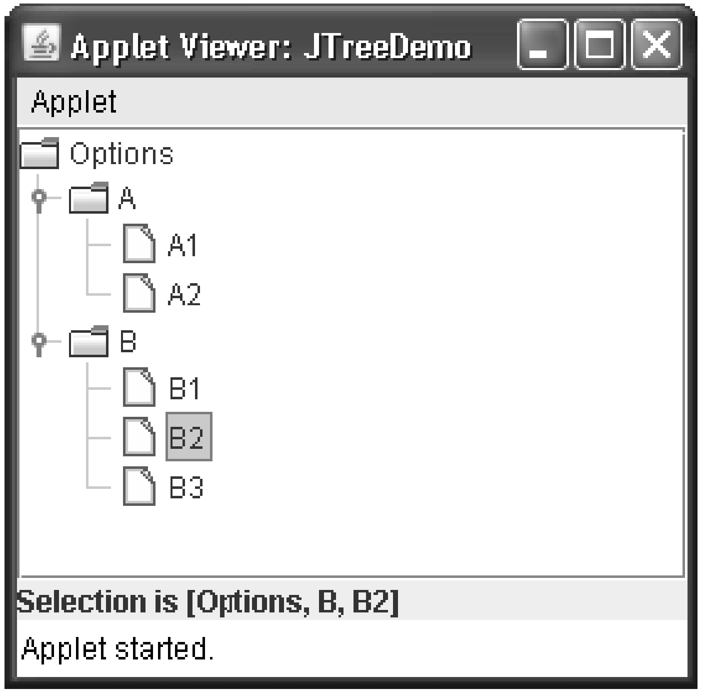
```
import java.awt.*;
import javax.swing.event.*;
import javax.swing.*;
import javax.swing.tree.*;
public class JTreeDemo extends JApplet {
    JTree tree;
    JLabel jlab;
    public void init() {
      try {
            SwingUtilities.invokeAndWait( new Runnable() {
                        public void run() {
                        makeGUI();
                        }
                  } );
      } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
      }
    }
```

JTree

```
private void makeGUI() {
        // Create top node of tree.
        DefaultMutableTreeNode top = new DefaultMutableTreeNode("Options");

        // Create subtree of "A".
        DefaultMutableTreeNode a = new DefaultMutableTreeNode("A");
        top.add(a);
        DefaultMutableTreeNode a1 = new DefaultMutableTreeNode("A1");
        a.add(a1);
        DefaultMutableTreeNode a2 = new DefaultMutableTreeNode("A2");
        a.add(a2);

        // Create subtree of "B".
        DefaultMutableTreeNode b = new DefaultMutableTreeNode("B");
        top.add(b);
        DefaultMutableTreeNode b1 = new DefaultMutableTreeNode("B1");
        b.add(b1);
        DefaultMutableTreeNode b2 = new DefaultMutableTreeNode("B2");
        b.add(b2);
        DefaultMutableTreeNode b3 = new DefaultMutableTreeNode("B3");
        b.add(b3);
```

JTree

```
        // Create the tree.
        tree = new JTree(top);

         // Add the tree to a scroll pane.
        JScrollPane jsp = new JScrollPane(tree);

        // Add the scroll pane to the content pane.
        add(jsp);

        // Add the label to the content pane.
        jlab = new JLabel();
        add(jlab, BorderLayout.SOUTH);

        // Handle tree selection events.
        tree.addTreeSelectionListener(new TreeSelectionListener() {
                public void valueChanged(TreeSelectionEvent tse) {
                        jlab.setText("Selection is " + tse.getPath());
                }
        });
    }
}
```



```
Selection is [Options, B, B2]
Applet started.
```

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.
- Depending on its configuration, it is also possible to select a row, column, or cell within the table, and to change the data within a cell.
- **JTable** 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

- 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.

JTable Example

- The following example illustrates how to create and use a simple table. A one-dimensional array of strings called **colHeads** is created for the column headings. A two-dimensional array of strings called **data** is created for the table cells. You can see that each element in the array is an array of three strings. These arrays are passed to the **JTable** constructor. The table is added to a scroll pane, and then the scroll pane is added to the content pane. The table displays the data in the **data** array. The default table configuration also allows the contents of a cell to be edited. Changes affect the underlying array, which is **data** in this case.

JTable

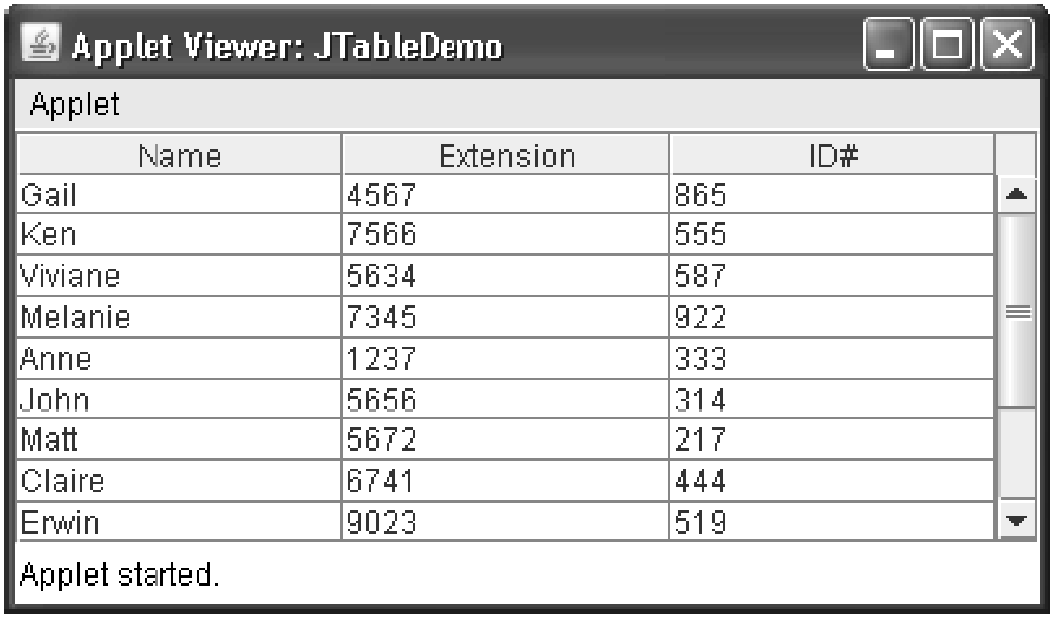
```
import java.awt.*;
import javax.swing.*;
public class JTableDemo extends JApplet {
    public void init() {
    try {
            SwingUtilities.invokeAndWait( new Runnable() {
                        public void run() {
                            makeGUI();
                        }
                  } );
    } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
    }
    }
```

JTable

```
private void makeGUI() {
        // Initialize column headings.
        String[] colHeads = { "Name", "Extension", "ID#" };

        // Initialize data.
        Object[][] data = {
                { "Gail", "4567", "865" },
                { "Ken", "7566", "555" },
                { "Viviane", "5634", "587" },
                { "Melanie", "7345", "922" },
                { "Anne", "1237", "333" },
                { "John", "5656", "314" },
                { "Matt", "5672", "217" },
                { "Claire", "6741", "444" },
                { "Erwin", "9023", "519" },
                { "Ellen", "1134", "532" },
                { "Jennifer", "5689", "112" },
                { "Ed", "9030", "133" },
                { "Helen", "6751", "145" }
        };
```

JTable

```
    // Create the table.
    JTable table = new JTable(data, colHeads);

    // Add the table to a scroll pane.
    JScrollPane jsp = new JScrollPane(table);

    // Add the scroll pane to the content pane.
    add(jsp);
    }
}
```



Name	Extension	ID#
Gail	4567	865
Ken	7566	555
Viviane	5634	587
Melanie	7345	922
Anne	1237	333
John	5656	314
Matt	5672	217
Claire	6741	444
Erwin	9023	519

```
Applet started.
```