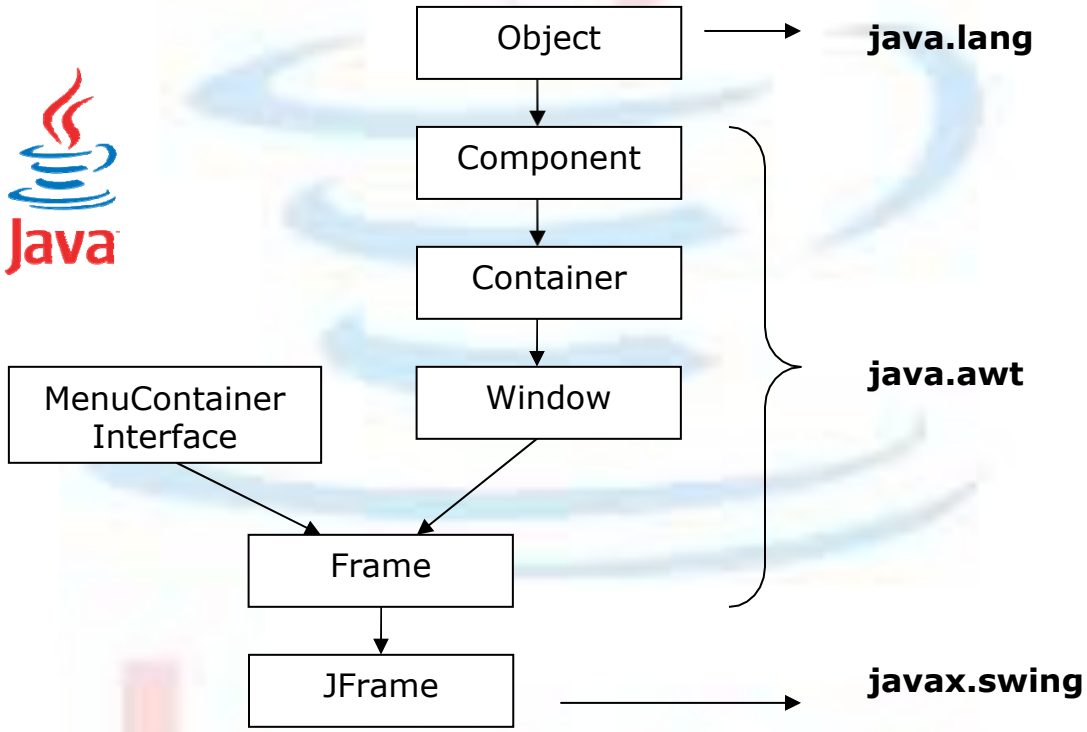
Chapter 02
The Tour of Swing

Contents:

Introduction

Swing is a set of classes this provides more powerful and flexible components than are possible with the AWT. In addition to the familiar components, such as buttons, check boxes, and labels, Swing supplies several exciting additions, including tabbed panes, scroll panes, trees, and tables. Even familiar components such as buttons have more capabilities in Swing. For example, a button may have both an image and a text string associated with it. Also, the image can be changed as the state of the button changes.

Unlike AWT components, Swing components are not implemented by platform-specific code. Instead, they are written entirely in Java and, therefore, are platform-independent. The term lightweight is used to describe such elements. The number of classes and interfaces in the Swing packages is substantial. Swing is the set of packages built on top of the AWT that provide us with a great number of pre-built classes that is, over 250 classes and 40 UI components.

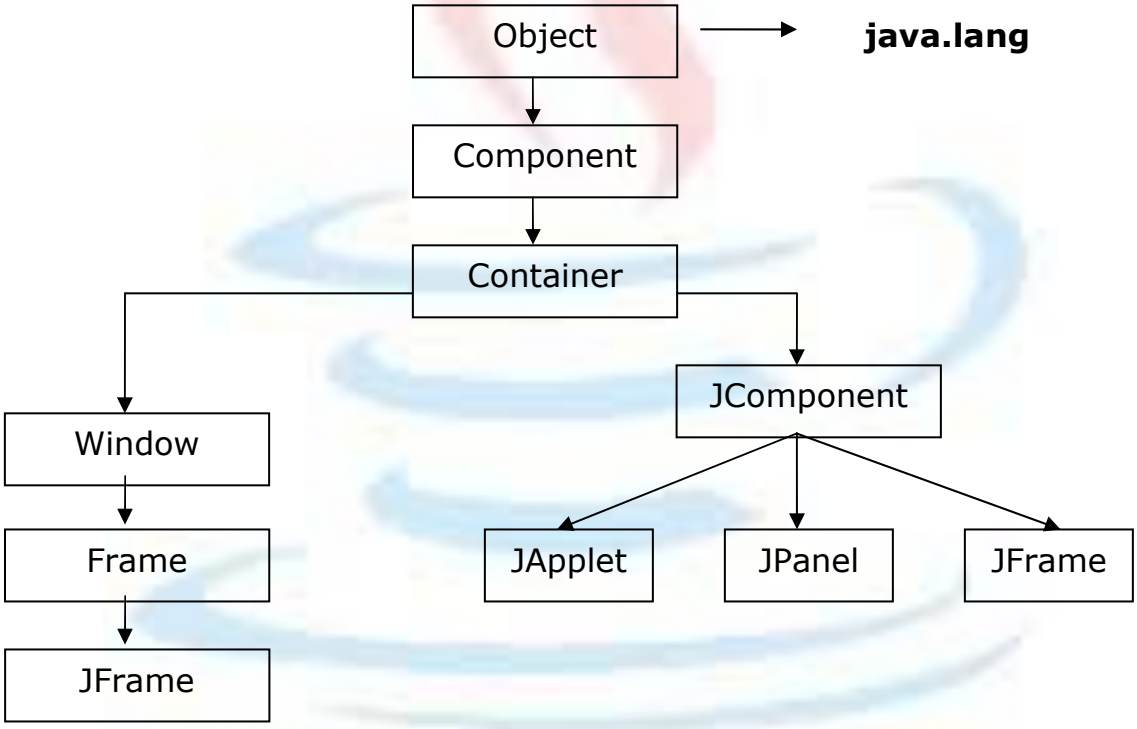


The Swing component classes that are shown below:

Class	Description
AbstractButton	Abstract super-class for Swing buttons.
ButtonGroup	Encapsulates a mutually exclusive set of buttons.
ImageIcon	Encapsulates an icon.
JApplet	The Swing version of Applet.
JButton	The Swing push button class.
JCheckBox	The Swing check box class.

JComboBox	Encapsulates a combo box (a combination of a drop-down list and text field).
JLabel	The Swing version of a label.
JRadioButton	The Swing version of a radio button.
JScrollPane	Encapsulates a scrollable window.
JTabbedPane	Encapsulates a tabbed window.
JTable	Encapsulates a table-based control.
JTextField	The Swing version of a text field.
JTree	Encapsulates a tree-based control.

The Swing-related classes are contained in javax.swing and its subpackages, such as javax.swing.tree.



The Swing family tree (Ref. No. 2)

Swing Features *(Ref. No. 2)*

Besides the large array of components in Swing and the fact that they are lightweight, Swing introduces many other innovations.

2.2.1 Borders

We can draw borders in many different styles around components using the setborder( ) method.

2.2.2 Graphics Debugging

We can use setDebuggingGraphicsOptions method to set up graphics debugging which means among the other things, that you can watch each line as its drawn and make it flash.

2.2.3 Easy mouseless operation

It is easy to connect keystrokes to components.

2.2.4 Tooltips

We can use the setToolTipText method of JComponent to give components a tooltip, one of those small windows that appear when the mouse hovers over a component and gives explanatory text.

2.2.5 Easy Scrolling

We can connect scrolling to various components-something that was impossible in AWT.

2.2.6 Pluggable look and feel

We can set the appearance of applets and applications to one of three standard looks. Windows, Motif (Unix) or Metal (Standard swing look).

2.2.7 New Layout Managers

Swing introduces the BoxLayout and OverlayLayout layout managers.

One of the differences between the AWT and Swing is that, when we redraw items on the screen of AWT, the update method is called first to redraw the item's background and programmers often override update method to just call the paint method directly to avoid flickering. In Swing, on the other hand the update method does not redraw the item's background because components can be transparent; instead update just calls paint method directly.

JApplet

Fundamental to Swing is the JApplet class, which extends Applet. Applets that use Swing must be subclasses of JApplet. JApplet is rich with functionality that is not found in Applet. For example, JApplet supports various "panes," such as the content pane, the glass pane, and the root pane.

When adding a component to an instance of JApplet, do not invoke the add( ) method of the applet. Instead, call add( ) for the content pane of the JApplet object. The content pane can be obtained via the method shown here:

```
Container getContentPane( )
```

The add( ) method of Container can be used to add a component to a content pane. Its form is shown here:

```
void add(comp)
```

Here, *comp* is the component to be added to the content pane.

Icons and Labels

In Swing, icons are encapsulated by the ImageIcon class, which paints an icon from an image. Two of its constructors are shown here:

```
ImageIcon(String filename)
ImageIcon(URL url)
```

The first form uses the image in the file named *filename*. The second form uses the image in the resource identified by *url*. The ImageIcon class implements the Icon interface that declares the methods shown here:

Method	**Description**
int getIconHeight( )	Returns the height of the icon in pixels.
int getIconWidth( )	Returns the width of the icon in pixels.
void paintIcon(Component comp, Graphics g, int x, int y)	Paints the icon at position x,y on the graphics context g. Additional information about the paint operation can be provided in comp.

Swing labels are instances of the JLabel class, which extends JComponent. It can display text and/or an icon. Some of its constructors are shown here:

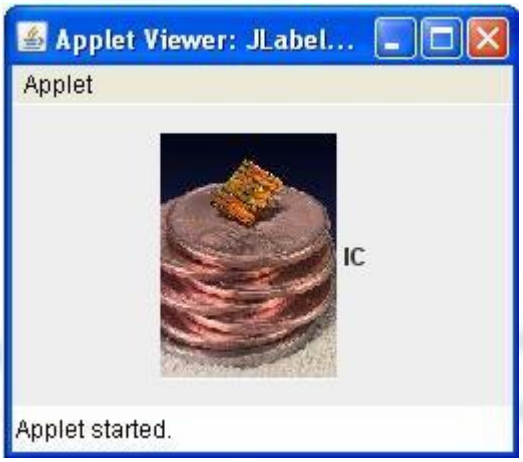
```
JLabel(Icon i)
Label(String s)
JLabel(String s, Icon i, int align)
```

Here, *s* and *i* are the text and icon used for the label. The *align* argument is either LEFT, RIGHT,CENTER, LEADING, or TRAILING. These constants are defined in the SwingConstants interface, along with several others used by the Swing classes. The icon and text associated with the label can be read and written by the following methods:

```
Icon getIcon(  )
String getText(  )
void setIcon(Icon i)
void setText(String s)
```

Here, *i* and *s* are the icon and text, respectively. The following example illustrates how to create and display a label containing both an icon and a string. The applet begins by getting its content pane. Next, an ImageIcon object is created for the file IC.jpg. This is used as the second argument to the JLabel constructor. The first and last arguments for the JLabel constructor are the label text and the alignment. Finally, the label is added to the content pane.

```
import java.awt.*;
import javax.swing.*;
/* <applet code="JLabelDemo" width=250 height=150> </applet> */
public class JLabelDemo extends JApplet
{
        public void init()
        {
                Container contentPane = getContentPane();
                ImageIcon ii = new ImageIcon("IC.jpg");
                JLabel jl = new JLabel("IC", ii, JLabel.CENTER);
                contentPane.add(jl);
        }
}
```



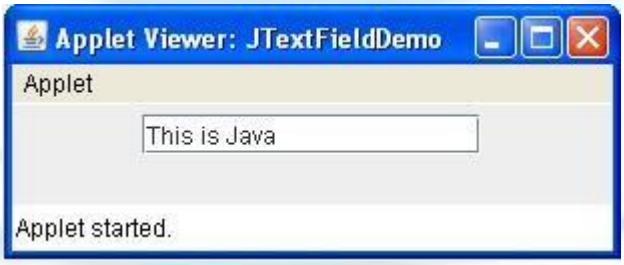
Text Fields

The Swing text field is encapsulated by the JTextComponent class, which extends JComponent. It provides functionality that is common to Swing text components. One of its subclasses is JTextField, which allows us to edit one line of text. Some of its constructors are shown here:

```
JTextField( )
JTextField(int cols)
JTextField(String s, int cols)
JTextField(String s)
```

Here, *s* is the string to be presented, and *cols* is the number of columns in the text field. The following example illustrates how to create a text field. The applet begins by getting its content pane, and then a flow layout is assigned as its layout manager. Next, a JTextField object is created and is added to the content pane.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTextFieldDemo" width=300 height=50>
</applet>
*/
public class JTextFieldDemo extends JApplet
{
    JTextField jtf;
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        jtf = new JTextField(15);
        contentPane.add(jtf);
    }
}
```



Buttons

Swing buttons provide features that are not found in the Button class defined by the AWT. For example, we can associate an icon with a Swing button. Swing buttons are subclasses of the AbstractButton class, which extends JComponent. AbstractButton contains many methods that allow us to control the behavior of buttons, check box and radio buttons. For example, we can define different icons that are displayed for the component when it is disabled, pressed, or selected. Another icon can be used as rollover icon, which is displayed when the mouse is positioned over that component. The following are the methods that control this behavior:

```
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 these different conditions. The text associated with a button can be read and written via the following methods:

```
String getText(  )
void setText(String s)
```

Here, *s* is the text to be associated with the button.

Concrete subclasses of AbstractButton generate action events when they are pressed. Listeners register and un-register for these events via the methods shown here:

```
void addActionListener(ActionListener al)
void removeActionListener(ActionListener al)
```

Here, *al* is the action listener. AbstractButton is a superclass for push buttons, check boxes, and radio buttons.

JButton Class

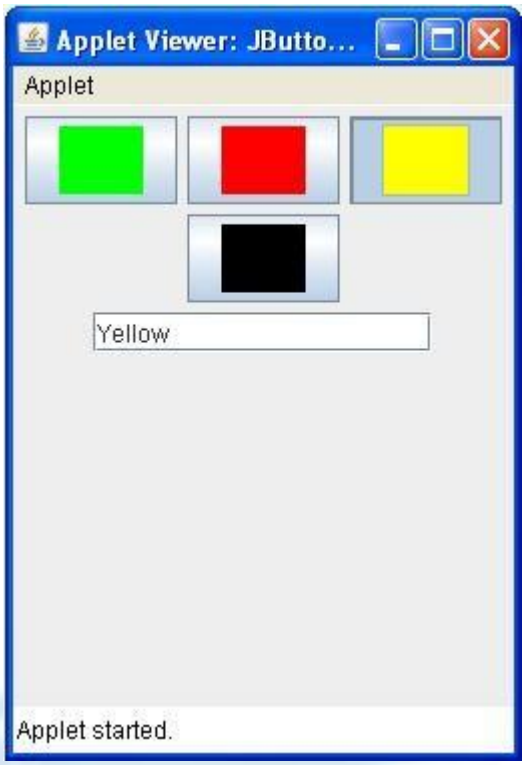
The JButton class provides the functionality of a push button. JButton allows an icon string, or both to be associated with the push button. Some of its constructors are shown here:

```
JButton(Icon i)
JButton(String s)
JButton(String s, Icon i)
```

Here, *s* and *i* are the string and icon used for the button. The following example displays four push buttons and a text field. 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 text field. The applet begins by getting its content pane and setting the layout manager of that pane. Four image buttons are created and added to the content pane. Next, the applet is registered to receive action events that are generated by the buttons. A text field is then created and added to the applet. Finally, a handler for action events displays the command string that is associated with the button. The text field is used to present this string.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JButtonDemo" width=250 height=300>
```

```
</applet>
*/
public class JButtonDemo extends JApplet
implements ActionListener
{
    JTextField jtf;
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        ImageIcon france = new ImageIcon("green.jpg");
        JButton jb = new JButton(france);
        jb.setActionCommand("Green");
        jb.addActionListener(this);
        contentPane.add(jb);
        ImageIcon germany = new ImageIcon("red.jpg");
        jb = new JButton(germany);
        jb.setActionCommand("Red");
        jb.addActionListener(this);
        contentPane.add(jb);
        ImageIcon italy = new ImageIcon("yellow.jpg");
        jb = new JButton(italy);
        jb.setActionCommand("Yellow");
        jb.addActionListener(this);
        contentPane.add(jb);
        ImageIcon japan = new ImageIcon("black.jpg");
        jb = new JButton(japan);
        jb.setActionCommand("Black");
        jb.addActionListener(this);
        contentPane.add(jb);
        jtf = new JTextField(15);
        contentPane.add(jtf);
    }
    public void actionPerformed(ActionEvent ae)
    {
        jtf.setText(ae.getActionCommand());
    }
}
```



Check Boxes

The **JCheckBox** class, which provides the functionality of a check box, is a concrete implementation of AbstractButton. **It is immediate super-class is JToggleButton, which provides support for two-state buttons**. Some of its constructors are shown here:

```
JCheckBox(Icon i)
JCheckBox(Icon i, boolean state)
JCheckBox(String s)
JCheckBox(String s, boolean state)
JCheckBox(String s, Icon i)
JCheckBox(String s, Icon i, boolean state)
```

Here, *i* is the icon for the button. The text is specified by *s*. If *state* is true, the check box is initially selected. Otherwise, it is not. The state of the check box can be changed via the following method:

```
void setSelected(boolean state)
```

Here, *state* is true if the check box should be checked. The following example illustrates how to create an applet that displays four check boxes and a text field. When a check box is pressed, its text is displayed in the text field the content pane for the JApplet object is obtained, and a flow layout is assigned as its layout manager. Next, four check boxes are added to the

content pane, and icons are assigned for the normal, rollover, and selected states. The applet is then registered to receive item events. Finally, a text field is added to the content pane. When a check box is selected or deselected, an item event is generated. This is handled by itemStateChanged( ). Inside itemStateChanged( ), the getItem( ) method gets the JCheckBox object that generated the event. The getText( ) method gets the tex for that check box and uses it to set the text inside the text field.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JCheckBoxDemo" width=400 height=50>
</applet>
*/
public class JCheckBoxDemo extends JApplet
implements ItemListener
{
    JTextField jtf;
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        JCheckBox cb = new JCheckBox("C", true);
        cb.addItemListener(this);
        contentPane.add(cb);
        cb = new JCheckBox("C++", false);
        cb.addItemListener(this);
        contentPane.add(cb);
        cb = new JCheckBox("Java", false);
        cb.addItemListener(this);
        contentPane.add(cb);
        cb = new JCheckBox("Perl", false);
        cb.addItemListener(this);
        contentPane.add(cb);
        jtf = new JTextField(15);
        contentPane.add(jtf);
    }
    public void itemStateChanged(ItemEvent ie)
    {
        JCheckBox cb = (JCheckBox)ie.getItem();
        jtf.setText(cb.getText());
    }
}
```

Radio Buttons

Radio buttons are supported by the JRadioButton class, which is a concrete implementation of AbstractButton. Its immediate super-class is JToggleButton, which provides support for two-state buttons. Some of its constructors are shown here:

```
JRadioButton(Icon i)
JRadioButton(Icon i, boolean state)
JRadioButton(String s)
JRadioButton(String s, boolean state)
JRadioButton(String s, Icon i)
JRadioButton(String s, Icon i, boolean state)
```

Here, *i* is the icon for the button. The text is specified by s. If *state* istrue, the button is initially selected. Otherwise, it is not. Radio buttons must be configured into a group. Only one of the buttons in that 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.

The ButtonGroup class is instantiated to create a button group. Its default constructor is invoked for this purpose. Elements are then added to the button group via the following method:

```
void add(AbstractButton ab)
```

Here, *ab* is a reference to the button to be added to the group. The following example illustrates how to use radio buttons. Three radio buttons and one text field are created. When a radio button is pressed, its text is displayed in the text field. First, the content pane for the JApplet object is obtained and a flow layout is assigned as its layout manager. Next, three radio buttons are added to the content pane. Then, a button group is defined and the buttons are added to it. Finally, a text field is added to the content pane.

Radio button presses generate action events that are handled by actionPerformed(). The getActionCommand() method gets the text that is associated with a radio button and uses it to set the text field.

```
import java.awt.*;
```

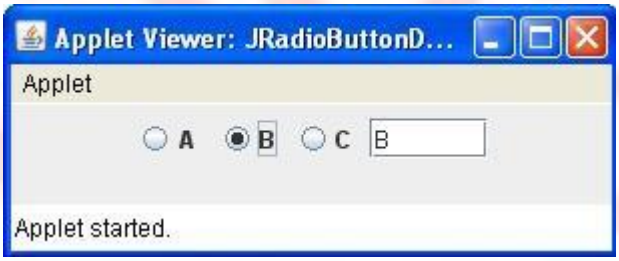
```
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JRadioButtonDemo" width=300 height=50>
</applet>
*/
public class JRadioButtonDemo extends JApplet
implements ActionListener
{
    JTextField tf;
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());

        JRadioButton b1 = new JRadioButton("A");
        b1.addActionListener(this);
        contentPane.add(b1);

        JRadioButton b2 = new JRadioButton("B");
        b2.addActionListener(this);
        contentPane.add(b2);

        JRadioButton b3 = new JRadioButton("C");
        b3.addActionListener(this);
        contentPane.add(b3);

        ButtonGroup bg = new ButtonGroup();
        bg.add(b1);
        bg.add(b2);
        bg.add(b3);
        tf = new JTextField(5);
        contentPane.add(tf);
    }
    public void actionPerformed(ActionEvent ae)
    {
        tf.setText(ae.getActionCommand());
    }
}
```



Combo Boxes

Swing provides a combo box (a combination of a text field and a drop-down list) through the JComboBox class, which extends JComponent. A combo box normally displays one entry. However, it can also display a drop-down list that allows a user to select a different entry. We can also type our selection into the text field. Two of JComboBox's constructors are shown here:

```
JComboBox( )
JComboBox(Vector v)
JComboBox(Objects obj[])
```

Here, *v* is a vector that initializes the combo box and *obj* is the array of objects. Items are added to the list of choices via the addItem( ) method, whose signature is shown here:

```
void addItem(Object obj)
```

Here, *obj* is the object to be added to the combo box.

Important Methods:

`public void setEditable(boolean aFlag)`
It determines whether the JComboBox field is editable or not?

`public boolean isEditable()`
It returns true if the JComboBox is editable. By default, a combo box is not editable.

`public void setMaximumRowCount(int count)`
It sets the maximum number of rows the JComboBox displays. If the number of objects in the model is greater than 'count', the combo box uses a scrollbar.

`public void setSelectedItem(Object anObject)`
It sets the selected item in the combo box display area to the object in the argument. If anObject is in the list, the display area shows anObject selected.

`public void insertItemAt(Object anObject, int index)`
It inserts an item 'anObject' into the item list at a given 'index'.

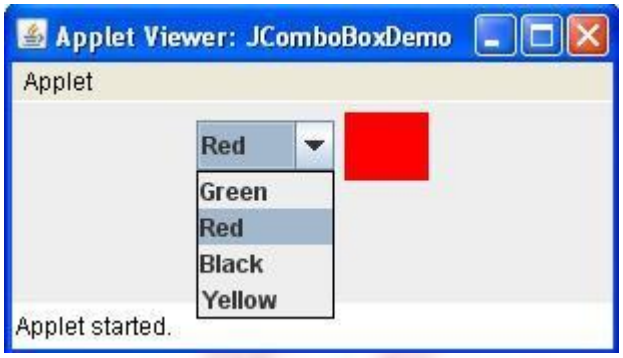
`public void removeItem(Object anObject)`
It removes an item 'anObject' from the item list.

```
public void removeItemAt(int anIndex)
```

It removes the item at 'anIndex'.

The following example contains a combo box and a label. The label displays an icon. The combo box contains entries for colors Green, Red, Yellow and Black. When a country is selected, the label is updated to display the color for that particular color. Color jpeg images are already stored in the current directory.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JComboBoxDemo" width=300 height=100>
</applet>
*/
public class JComboBoxDemo extends JApplet
implements ItemListener
{
    JLabel jl;
    ImageIcon green, red, black, yellow;
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        JComboBox jc = new JComboBox();
        jc.addItem("Green");
        jc.addItem("Red");
        jc.addItem("Black");
        jc.addItem("Yellow");
        jc.addItemListener(this);
        contentPane.add(jc);
        jl = new JLabel(new ImageIcon("green.jpg"));
        contentPane.add(jl);
    }
    public void itemStateChanged(ItemEvent ie)
    {
        String s = (String)ie.getItem();
        jl.setIcon(new ImageIcon(s + ".jpg"));
    }
}
```



Tabbed Panes

A tabbed pane is a component that appears as a group of folders in a file cabinet. Each folder has a title. When a user selects a folder, its contents become visible. Only one of the folders may be selected at a time. Tabbed panes are commonly used for setting configuration options.

Tabbed panes are encapsulated by the JTabbedPane class, which extends JComponent. There are three constructors of JTabbedPane.

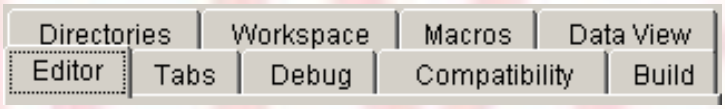
```
JTabbedPane()
JTabbedPane(int tabPlacement)
JTabbedPane(int tabPlacement, int tabLayoutPolicy)
```

The first form creates an empty TabbedPane with a default tab placement of JTabbedPane.TOP. Second form creates an empty TabbedPane with the specified tab placement of any of the following:

```
JTabbedPane.TOP
JTabbedPane.BOTTOM
JTabbedPane.LEFT
JTabbedPane.RIGHT
```

The third form of constructor creates an empty TabbedPane with the specified tab placement and tab layout policy. Tab placements are listed above. Tab layout policy may be either of the following:

```
JTabbedPane.WRAP_TAB_LAYOUT
JTabbedPane.SCROLL_TAB_LAYOUT
```



Wrap tab policy

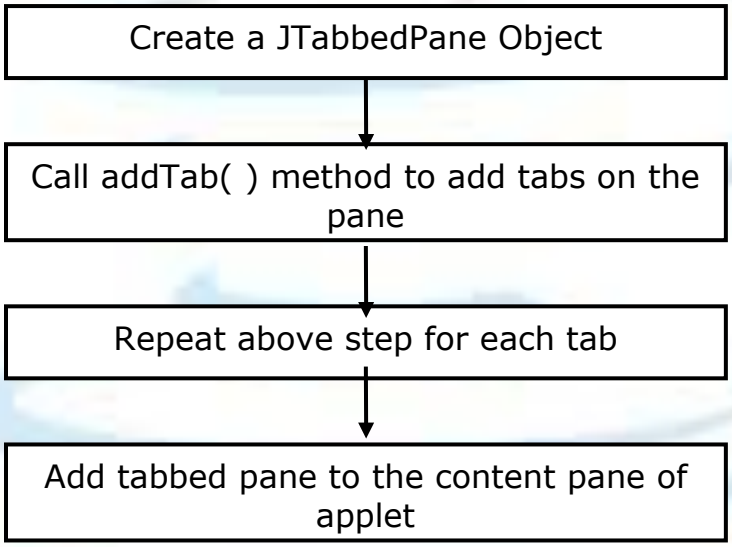
Scroll Tab Policy

Tabs are defined via the following method:

```
void addTab(String str, Component comp)
```

Here, *str* is the title for the tab, and comp is the component that should be added to the tab. Typically, a JPanel or a subclass of it is added. The general procedure to use a tabbed pane in an applet is outlined here:

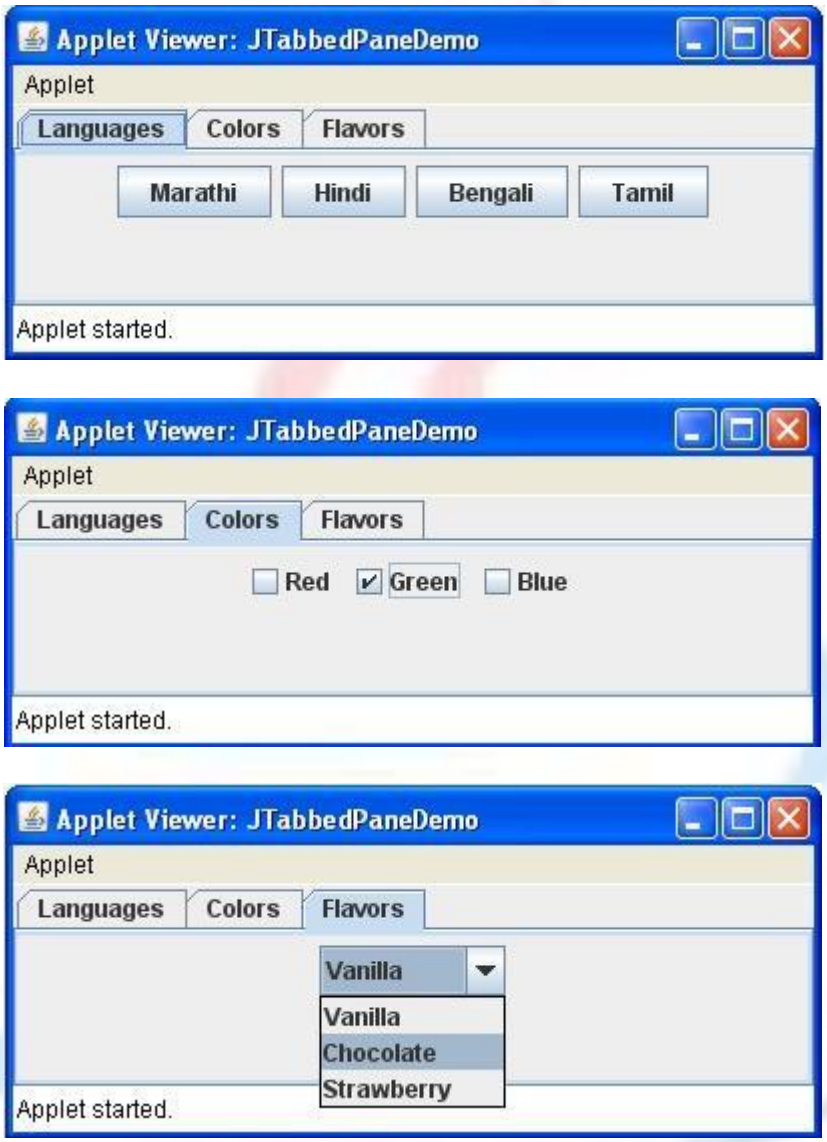
1. Create a JTabbedPane object.
2. Call addTab( ) to add a tab to the pane. (The arguments to this method define the title of the tab and the component it contains.)
3. Repeat step 2 for each tab.
4. Add the tabbed pane to the content pane of the applet.



The following example illustrates how to create a tabbed pane. The first tab is titled Languages and contains four buttons. Each button displays the name of a language. 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.

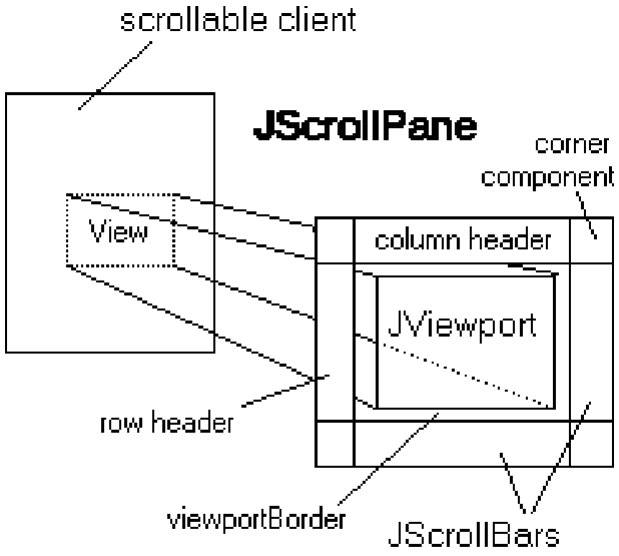
```
import javax.swing.*;
/*
<applet code="JTabbedPaneDemo" width=400 height=100>
</applet>
*/
```

```
public class JTabbedPaneDemo extends JApplet
{
    public void init()
    {
        JTabbedPane jtp = new JTabbedPane();
        jtp.addTab("Languages", new LangPanel());
        jtp.addTab("Colors", new ColorsPanel());
        jtp.addTab("Flavors", new FlavorsPanel());
        getContentPane().add(jtp);
    }
}
class LangPanel extends JPanel
{
    public LangPanel()
    {
        JButton b1 = new JButton("Marathi");
        add(b1);
        JButton b2 = new JButton("Hindi");
        add(b2);
        JButton b3 = new JButton("Bengali");
        add(b3);
        JButton b4 = new JButton("Tamil");
        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);
    }
}
```



Scroll Panes

A scroll pane is a component that presents a rectangular area in which a component may be viewed. Horizontal and/or vertical scroll bars may be provided if necessary. Scroll panes are implemented in Swing by the JScrollPane class, which extends JComponent.



Some of its constructors are shown here:

```
JScrollPane()
JScrollPane(Component comp)
JScrollPane(int vsb, int hsb)
JScrollPane(Component comp, int vsb, int hsb)
```

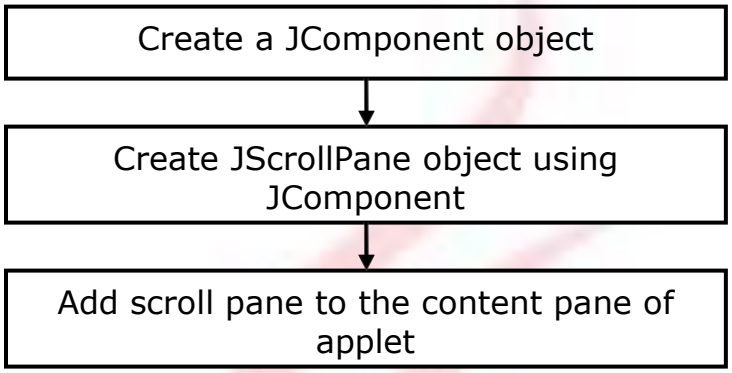
Here, *comp* is the component to be added to the scroll pane. *vsb* and *hsb* are int constants that define when vertical and horizontal scroll bars for this scroll pane are shown. These constants are defined by the **ScrollPaneConstants** interface. Some examples of these constants are described as follows:

Constant	**Description**
HORIZONTAL_SCROLLBAR_ALWAYS	Always provide horizontal scroll bar
HORIZONTAL_SCROLLBAR_AS_NEEDED	Provide horizontal scroll bar, if needed
VERTICAL_SCROLLBAR_ALWAYS	Always provide vertical scroll bar
VERTICAL_SCROLLBAR_AS_NEEDED	Provide vertical scroll bar, if needed

Here are the steps that you should follow to use a scroll pane in an applet:

1. Create a JComponent object.
2. Create a JScrollPane object. (The arguments to the constructor specify the component and the policies for vertical and horizontal scroll bars.)

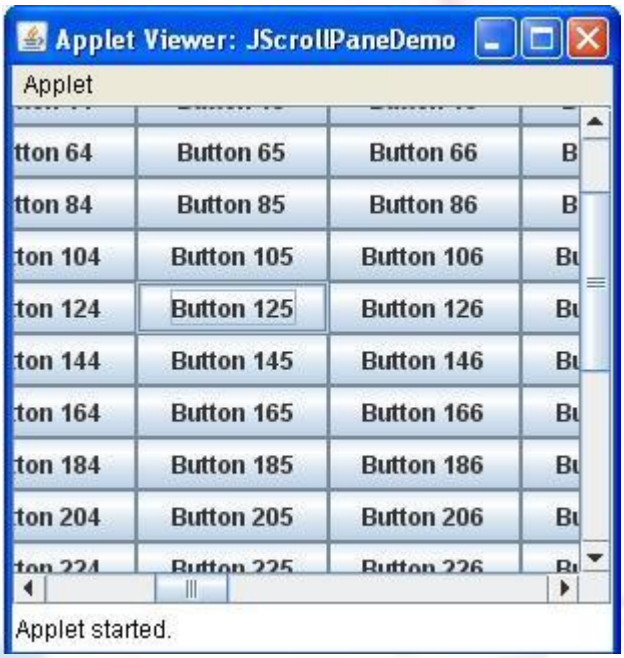
3. Add the scroll pane to the content pane of the applet.



The following example illustrates a scroll pane. First, the content pane of the JApplet object is obtained and a border layout is assigned as its layout manager. Next, a JPanel object is created and four hundred buttons are added to it, arranged into twenty columns. The panel is then added to a scroll pane, and the scroll pane is added to the content pane. This causes vertical and horizontal scroll bars to appear. We can use the scroll bars to scroll the buttons into view.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JScrollPaneDemo" width=300 height=250>
</applet>
*/
public class JScrollPaneDemo extends JApplet
{
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());
        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;
            }
        }
        int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
        int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
        JScrollPane jsp = new JScrollPane(jp, v, h);
        contentPane.add(jsp, BorderLayout.CENTER);
    }
```

}



Trees

A tree is a component that presents a hierarchical view of data. A user has the ability to expand or collapse individual sub-trees in this display.

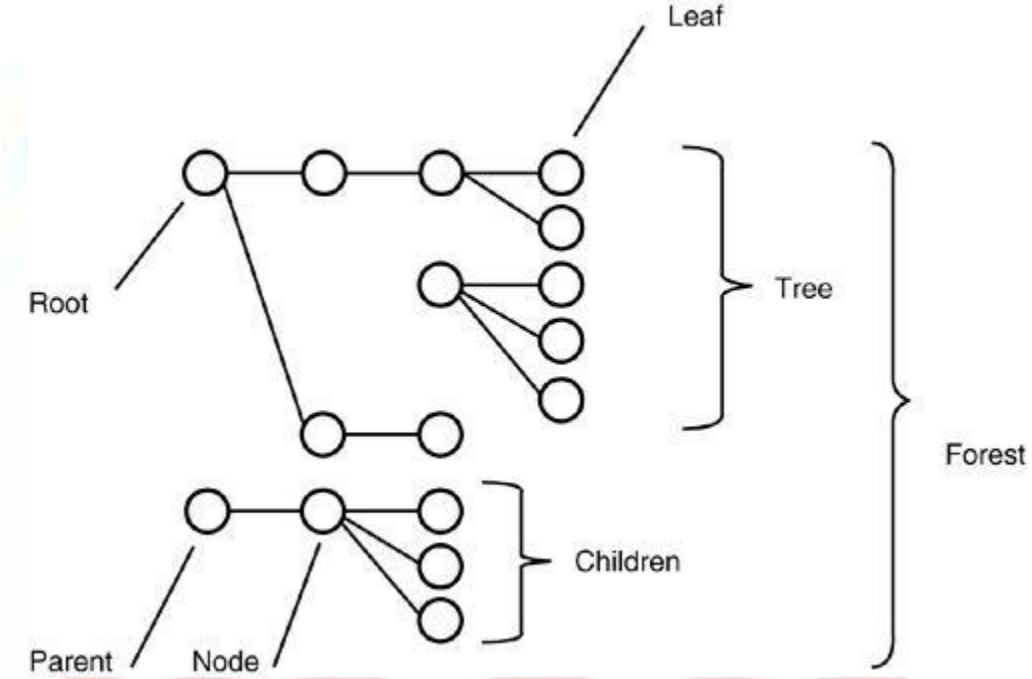


Fig. Tree Terminology

Trees are implemented in Swing by the JTree class, which extends JComponent. Some of its constructors are shown here:

```
JTree(Hashtable ht)
JTree(Object obj[ ])
JTree(TreeNode tn)
JTree(Vector v)
```

The first form creates a tree in which each element of the hash table *ht* is a child node. Each element of the array *obj* is a child node in the second form. The tree node *tn* is the root of the tree in the third form. Finally, the last form uses the elements of vector *v* as child nodes. A JTree object generates events when a node is expanded or collapsed. The addTreeExpansionListener( ) and removeTreeExpansionListener( ) methods allow listeners to register and unregister for these notifications. The signatures of these methods are shown here:

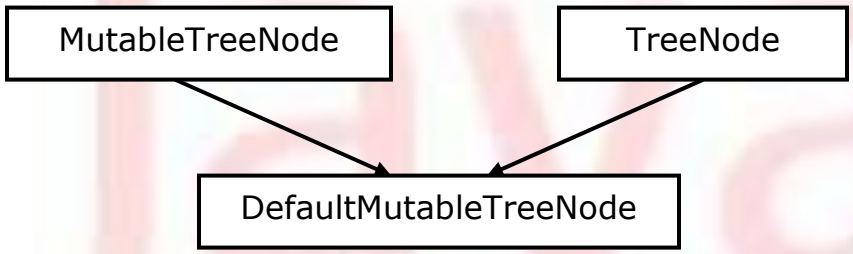
```
void addTreeExpansionListener(TreeExpansionListener tel)
void removeTreeExpansionListener(TreeExpansionListener tel)
```

Here, *tel* is the listener object. The getPathForLocation( ) method is used to translate a mouse click on a specific point of the tree to a tree path. Its signature is shown here:

```
TreePath getPathForLocation(int x, int y)
```

Here, *x* and *y* are the coordinates at which the mouse is clicked. The return value is a TreePath object that encapsulates information about the tree node that was selected by the user. The TreePath class encapsulates information about a path to a particular node in a tree. It provides several constructors and methods.

The TreeNode interface declares methods that obtain information about a tree node. For example, it is possible to obtain a reference to the parent node or an enumeration of the child nodes. The MutableTreeNode interface extends TreeNode. It declares methods that can insert and remove child nodes or change the parent node.



The DefaultMutableTreeNode class implements the MutableTreeNode interface. It represents a node in a tree. One of its constructors is shown here:

```
DefaultMutableTreeNode(Object obj)
```

Here, *obj* is the object to be enclosed in this tree node. The new tree node doesn't have a parent or children. To create a hierarchy of tree nodes, the add( ) method of DefaultMutableTreeNode can be used. Its signature is shown here:

```
void add(MutableTreeNode child)
```

Here, child is a mutable tree node that is to be added as a child to the current node.

Tree expansion events are described by the class TreeExpansionEvent in the javax.swing.event package. The getPath( ) method of this class returns a TreePath object that describes the path to the changed node. Its signature is shown here:

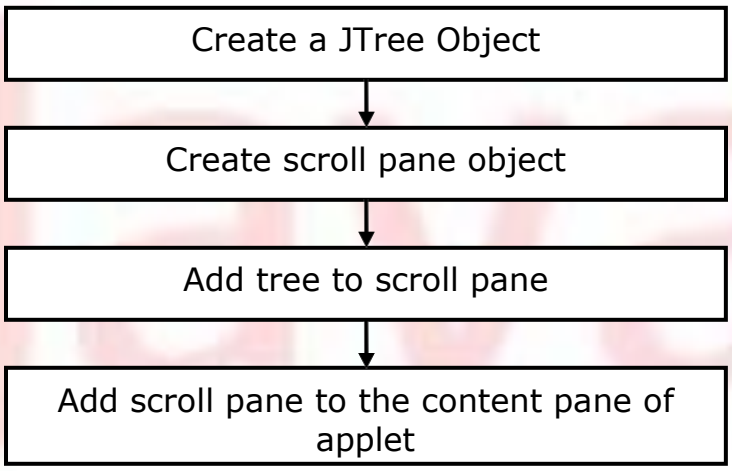
```
TreePath getPath( )
```

The TreeExpansionListener interface provides the following two methods:

```
void treeCollapsed(TreeExpansionEvent tee)
void treeExpanded(TreeExpansionEvent tee)
```

Here, *tee* is the tree expansion event. The first method is called when a sub-tree is hidden, and the second method is called when a sub-tree becomes visible. Here are the steps that we should follow to use a tree in an applet:

1. Create a JTree object.
2. Create a JScrollPane object. (The arguments to the constructor specify the tree and the policies for vertical and horizontal scroll bars.)
3. Add the tree to the scroll pane.
4. Add the scroll pane to the content pane of the applet.

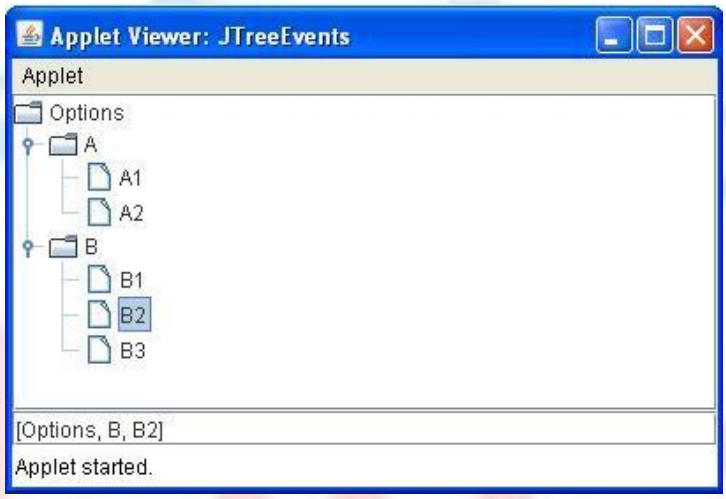


The following example illustrates how to create a tree and recognize mouse clicks on it. The init( ) method gets the content pane for the applet. A DefaultMutableTreeNode object labeled Options is created. This is the top node of the tree hierarchy. Additional tree nodes are then created, and the add( ) 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 applet. Next, a text field is created and added to the applet. Information about mouse click events is presented in this text field. To receive mouse events from the tree, the addMouseListener( ) method of the JTree object is called. The argument to this method is an anonymous inner class that extends MouseAdapter and overrides the mouseClicked( ) method.

The doMouseClicked( ) method processes mouse clicks. It calls getPathForLocation( ) to translate the coordinates of the mouse click into a TreePath object. If the mouse is clicked at a point that does not cause a node selection, the return value from this method is null. Otherwise, the tree path can be converted to a string and presented in the text field.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.tree.*;
/*
<applet code="JTreeEvents" width=400 height=200>
</applet>
*/
public class JTreeEvents extends JApplet
{
    JTree tree;
    JTextField jtf;
    public void init()
    {
        Container contentPane=getContentPane();
        contentPane.setLayout(new BorderLayout());
        DefaultMutableTreeNode top=new
               DefaultMutableTreeNode("Options");
        DefaultMutableTreeNode a= new DefaultMutableTreeNode("A");
        top.add(a);
        DefaultMutableTreeNode a1=new DefaultMutableTreeNode("A1");
        a.add(a1);
        DefaultMutableTreeNode a2=new DefaultMutableTreeNode("A2");
        a.add(a2);
        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);
        tree=new JTree(top);
        int v=ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
        int h=ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
        JScrollPane jsp=new JScrollPane(tree,v,h);
        contentPane.add(jsp,BorderLayout.CENTER);
        jtf=new JTextField("",20);
        contentPane.add(jtf,BorderLayout.SOUTH);
        tree.addMouseListener(new MouseAdapter()
        {
            public void mouseClicked(MouseEvent me)
            {
                doMouseClicked(me);
            }
        });
    }
    void doMouseClicked(MouseEvent me)
    {
        TreePath tp=tree.getPathForLocation(me.getX(),me.getY());
        if(tp!=null)
            jtf.setText(tp.toString());
        else
            jtf.setText("");
    }
}
```



The string presented in the text field describes the path from the top tree node to the selected node.

Tables

A table is a component that displays rows and columns of data. We can drag the cursor on column boundaries to resize columns. We can also drag a column to a new position. Tables are implemented by the JTable class, which extends JComponent. One of its constructors is shown here:

```
JTable(Object data[ ][ ], Object colHeads[ ])
JTable(int numRows, int numColumns)
JTable(Vector rowData, Vector columnData)
```

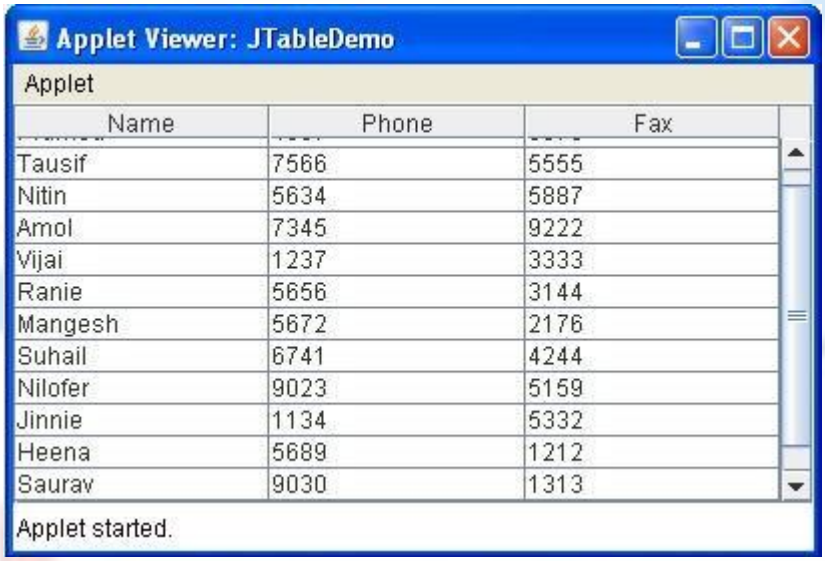
Here, *data* is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings. The *'numRows'* and *'numColumns'* are values with which the table is to be created. The *'rowData'* and *'columnData'* are the vector values by which the table is constructed.
Here are the steps for using a table in an applet:

1) Create a JTable object.
2) Create a JScrollPane object. (The arguments to the constructor specify the table and the policies for vertical and horizontal scroll bars.)
3) Add the table to the scroll pane.
4) Add the scroll pane to the content pane of the applet.

The following example illustrates how to create and use a table. The content pane of the JApplet object is obtained and a border layout is assigned as its layout manager. A one-dimensional array of strings is created for the column headings. This table has three columns. A two-dimensional array of strings is created for the table cells. We 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.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTableDemo" width=400 height=200>
</applet>
*/
public class JTableDemo extends JApplet
{
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());
        final String[] colHeads = { "Name", "Phone", "Fax" };
        final Object[][] data = {
```

```
            { "Pramod", "4567", "8675" },
            { "Tausif", "7566", "5555" },
            { "Nitin", "5634", "5887" },
            { "Amol", "7345", "9222" },
            { "Vijai", "1237", "3333" },
            { "Ranie", "5656", "3144" },
            { "Mangesh", "5672", "2176" },
            { "Suhail", "6741", "4244" },
            { "Nilofer", "9023", "5159" },
            { "Jinnie", "1134", "5332" },
            { "Heena", "5689", "1212" },
            { "Saurav", "9030", "1313" },
            { "Raman", "6751", "1415" }
    };
    JTable table = new JTable(data, colHeads);
    int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
    int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
    JScrollPane jsp = new JScrollPane(table, v, h);
    contentPane.add(jsp, BorderLayout.CENTER);
    }
}
```



References

1. **Java 2 the Complete Reference**,
   Fifth Edition by Herbert Schildt, 2001 Osborne McGraw Hill
   Chapter 26: The Tour of Swing
   (Most of the data is referred from this book)

2. **Java 6 Black Book,**
   Kogent Solutions, 2007, Dreamtech Press
   Chapter 15: Swing–Applets, Applications and Pluggable Look and Feel.

3. **JDK 5.0 Documentation,**
   Sun Microsystems, USA, www.java.sun.com
