* **HIntroduction to Swing**

Swing is a set of classes that 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.

Swing is actually part of a larger family of Java products known as the Java Foundation Classes ( JFC).

The Swing component classes that are used are shown here:

**Class Description**

AbstractButton Abstract superclass 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 (an 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**.

* **Swing Features:**

Swing provides many features for writing large-scale applications in Java.

1. **Pluggable Look-and-Feels:-**

One of the most exciting aspects of the Swing classes is the ability to dictate the L&F of each of the components, even resetting the L&F at runtime. L&Fs have become an important issue in GUI development over the past 10 years. Many users are familiar with the Motif style of user interface, which was common in Windows 3.1 and is still in wide use on Unix platforms.

Swing is capable of emulating several L&Fs and currently supports the Windows, Unix Motif, and "native" Java Metal L&Fs. Swing allows the user to switch L&Fs at runtime without having to close the application. This way, a user can experiment to see which L&F is best for her with instantaneous feedback.

****

1. **Lightweight Components:**

Most Swing components are lightweight. In the purest sense, this means that components are not dependent on native peers to render themselves. Instead, they use simplified graphics primitives to paint themselves on the screen and can even allow portions to be transparent.

AWT components are Heavyweight.

Heavyweight components were unwieldy for two reasons:

1. Equivalent components on different platforms don't necessarily act alike. A list component on one platform,for example, may work differently than a list component on another. Trying to coordinate and manage the differences between components was a formidable task.
2. The L&F of each component was tied to the host operating system and could not be changed.

With lightweight components, each component renders itself using the drawing primitives of theGraphics object (e.g., drawLine( ), fillRect( ), etc.). Lightweight components always render themselves onto the surface of the heavyweight top-level component they are contained in.

Almost all of the Swing components are lightweight; only a few top-level containers are not. This design allows programmers to draw (and redraw) the L&F of their application at runtime, instead of tying it to the L&F of the host operating system.

1. **Additional Features:**

Several other features distinguish Swing from the older AWT components.

* Swing has wide variety of new components, such as tables, trees, sliders, spinners, progress bars, internal frames, and text components.
* Swing components support the replacement of their insets with an arbitrary number of nested borders.
* Swing components can have *tooltips* placed over them. A tooltip is a textual pop up that momentarily appears when the mouse cursor rests inside the component's painting region. Tooltips can be used to give more information about the component in question.
* You can arbitrarily bind keyboard events to components, defining how they react to various keystrokes under given conditions.

**\*. The Model-View-Controller Architecture:**

Swing uses the*model-view-controller architecture* (MVC) as the fundamental design behind each of its components. Essentially, MVC breaks GUI components into three elements. Each of these elements plays a crucial role in how the component behaves.

***Model***

The model encompasses the state data for each component. There are different models for different types of components. For example, the model of a scrollbar component might contain information about the current position of its adjustable "thumb," its minimum and maximum values, and the thumb's width (relative to the range of values). A menu, on the other hand, may simply contain a list of the menu items the user can select from. This information remains the same no matter how the component is painted on the screen;model data is always independent of the component's visual representation.

***View***

The view refers to how you see the component on the screen. For a good example of how views can differ, look at an application window on two different GUI platforms. Almost all window frames have a title bar spanning the top of the window. However, the title bar may have a close box on the left side (like the Mac OS platform), or it may have the close box on the right side (as in the Windows platform). These are examples of different types of views for the same window object.

***Controller***

The controller is the portion of the user interface that dictates how the component interacts with events.Events come in many forms — e.g., a mouse click, gaining or losing focus, a keyboard event that triggers a specific menu command, or even a directive to repaint part of the screen. The controller decides how each component reacts to the event—if it reacts at all.

Figure shows how the model, view, and controller work together to create a scrollbar component. The scrollbar uses the information in the model to determine how far into the scrollbar to render the thumb and how wide the thumb should be. Note that the model specifies this information relative to the minimum and the maximum. It does not give the position or width of the thumb in screen pixels—the view calculates that. The view determines exactly where and how to draw the scrollbar, given the proportions offered by the model. The view knows whether it is a horizontal or vertical scrollbar, and it knows exactly how to shadow the end buttons and the thumb. Finally, the controller is responsible for handling mouse events on the component. The controller knows, for example, that dragging the thumb is a legitimate action for a scrollbar, within the limits defined by the endpoints, and that pushing on the end

buttons is acceptable as well. The result is a fully functional MVC scrollbar.

****

**MVC Interaction**

With MVC, each of the three elements—the model, the view, and the controller—requires the services of another element to keep itself continually updated. Let's continue discussing the scrollbar component.

We already know that the view cannot render the scrollbar correctly without obtaining information from the model first.

In this case, the scrollbar does not know where to draw its "thumb" unless it can obtain its current position and width relative to the minimum and maximum. Likewise, the view determines if the component is the recipient of user events, such as mouse clicks. (For example, the view knows the exact width of the thumb; it can tell whether a click occurred over the thumb or just outside of it.) The view passes these events on to the controller, which decides how to handle them. Based on the controller's decisions, the values in the model may need to be altered. If the user drags the scrollbar thumb, the controller reacts by incrementing the thumb's position in the model. At that point, the whole cycle repeats. The three elements, therefore, communicate their data as shown in Figure

**Figure . Communication through the model-view-controller architecture**

****

**MVC in Swing**

Swing actually uses a simplified variant of the MVC design called the *model-delegate* . This design combines the view and the controller object into a single element, the *UI delegate* , which draws the component to the screen and handles GUI events. Bundling graphics capabilities and event handling is somewhat easy in Java, since much of the event handling is taken care of in AWT. As you might expect, the communication between the model and the UIdelegate then becomes a two-way street, as shown in Figure

**Figure . With Swing, the view and the controller are combined into a UI-delegate object**

****

So let's review: each Swing component contains a model and a UI delegate. The model is responsible for maintaining information about the component's state. The UI delegate is responsible for maintaining information about how to draw the component on the screen. In addition, the UI delegate (in conjunction with AWT) reacts to various events that propagate through the component.

Note that the separation of the model and the UI delegate in the MVC design is extremely advantageous. One unique aspect of the MVC architecture is the ability to tie multiple views to a single model. For example, if you want to display the same data in a pie chart and in a table, you can base the views of two components on a single data model. That way, if the data needs to be changed, you can do so in only one place—the views update themselves accordingly . In the same manner, separating the delegate from the model gives the user the added benefit of choosing what a component looks like without affecting any of its data.

By using this approach, in conjunction with the lightweight design, Swing can provide each component with its own pluggable L&F.

***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*, Paints the icon at position *x*, *y* on the

int *x*, int *y*) 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**, or **CENTER**. 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

Example:

import java.awt.\*;

import javax.swing.\*;

/\*<applet code="swinglabel" width=400 height=400>

</applet>\*/

public class swinglabel extends JApplet

{

ImageIcon i=new ImageIcon("tabb.jpg");

JLabel l=new JLabel(i);

JLabel l1=new JLabel("OK");

JLabel l2=new JLabel("Cancel",i,JLabel.LEFT);

public void init()

{

Container c=getContentPane();

c.setLayout(new FlowLayout());

c.add(l);

c.add(l1);

c.add(l2);

}

}

**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 you 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() {

// Get content pane

Container contentPane = getContentPane();

contentPane.setLayout(new FlowLayout());

// Add text field to content pane

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, you 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 you to control the behavior of buttons, check boxes,

and radio buttons. For example, you can define different icons that are displayed for the

component 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 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 unregister 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. Each

is examined next.

**The JButton Class**

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. 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.

**Example:**

import java.awt.\*;

import javax.swing.\*;

import java.awt.event.\*;

/\*<applet code=jbut width=400 height=400>

</applet>\*/

public class jbut extends JApplet

{

JButton b;

ImageIcon i=new ImageIcon("Desert.jpg");

ImageIcon i1=new ImageIcon("Tulips.jpg");

public void init()

{

Container c=getContentPane();

b=new JButton(i);

//b.setPressedIcon(i1);

b.setRolloverIcon(i1);

b.setDisabledIcon(i1);

b.setEnabled(false);

c.add(b);

}

}

**Check Boxes**

The **JCheckBox** class, which provides the functionality of a check box, is a concrete

implementation of **AbstractButton**. 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.

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 text for that

check box and uses it to set the text inside the text field

**Example:**

import java.awt.\*;

import javax.swing.\*;

import java.awt.event.\*;

/\*<applet code=jcheck width=400 height=400>

</applet>\*/

public class jcheck extends JApplet

{

JCheckBox cb1,cb2;

ImageIcon i=new ImageIcon("Desert.jpg");

ImageIcon i1=new ImageIcon("Tulips.jpg");

public void init()

{

Container c=getContentPane();

c.setLayout(new FlowLayout());

cb1=new JCheckBox("Desert",i);

cb2=new JCheckBox("Tulip",i1,true);

//b.setPressedIcon(i1);

cb1.setSelectedIcon(i1);

cb2.setSelectedIcon(i);

c.add(cb1);

c.add(cb2);

}

}

**Radio Buttons**

Radio buttons are supported by the **JRadioButton** class, which is a concrete

implementation of **AbstractButton**. 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* is **true**, 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.

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.

**Example:**

import java.awt.\*;

import javax.swing.\*;

import java.awt.event.\*;

/\*<applet code=jradio width=400 height=400>

</applet>\*/

public class jradio extends JApplet

{

JRadioButton m,f;

ImageIcon i=new ImageIcon("Desert.jpg");

ImageIcon i1=new ImageIcon("Tulips.jpg");

ButtonGroup b=new ButtonGroup();

public void init()

{

Container c=getContentPane();

c.setLayout(new FlowLayout());

m=new JRadioButton("Male",i);

f=new JRadioButton("Female",i1);

b.add(m);

b.add(f);

c.add(m);

c.add(f);

}

}

**Combo Boxes**

Swing provides a *combo box* (a combination of a text field and a drop-down list) through

the 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. You can also type your selection into the text field. Two of **JComboBox**'s

constructors are shown here:

JComboBox( )

JComboBox(Vector v)

Here, *v* is a vector that initializes the combo box.

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.

**Example:**

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

import java.util.\*;

/\*

<applet code="JComboBoxDemo" width=300 height=100>

</applet>

\*/

public class JComboBoxDemo extends JApplet

{

ImageIcon d=new ImageIcon("Desert.jpg");

Vector v=new Vector(10);

public void init() {

Container contentPane = getContentPane();

contentPane.setLayout(new FlowLayout());

// Create a combo box and add it

// to the panel

v.addElement(10);

v.addElement("Hello");

JComboBox vc=new JComboBox(v);

JComboBox jc = new JComboBox();

jc.addItem(d);

jc.addItem("Second");

jc.addItem("Third");

jc.addItem("Fourth");

contentPane.add(vc);

contentPane.add(jc);

}

}

**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**. We will use its default constructor. Tabs are defined via the following

method:

void addTab(String str, Component comp)

void addTab(String str,Icon I,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. For each tab add a set of controls to a panel.then pass this panel object for addTab() method.

5. Add the tabbed pane to the content pane of the applet.

**Example:**

import java.awt.\*;

import javax.swing.\*;

/\*<applet code=jtabpane width=400 height=400>

</applet>\*/

public class jtabpane extends JApplet

{

JTabbedPane tb=new JTabbedPane();

JPanel p1,p2,p3;

JButton b1,b2,b3;

JLabel l1,l2,l3;

ImageIcon i=new ImageIcon("Desert.jpg");

ImageIcon i1=new ImageIcon("Tulips.jpg");

ImageIcon i2=new ImageIcon("Koala.jpg");

public void init()

{

Container c=getContentPane();

b1=new JButton(i);

b2=new JButton(i1);

b3=new JButton(i2);

l1=new JLabel("on panel1");

l2=new JLabel("on panel2");

l3=new JLabel("on panel3");

p1=new JPanel();

p2=new JPanel();

p3=new JPanel();

p1.add(l1);

p2.add(l2);

p3.add(l3);

p1.add(b1);

p2.add(b2);

p3.add(b3);

tb.addTab("p1",i,p1);

tb.addTab("p2",p2);

tb.addTab("p3",p3);

c.add(tb);

}

}

**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(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.

**Example:**

import java.awt.\*;

import javax.swing.\*;

/\*<applet code=jscroll width=400 height=400>

</applet>\*/

public class jscroll extends JApplet

{

JScrollPane js;

ImageIcon i=new ImageIcon("Koala.jpg");

JLabel l=new JLabel(i);

public void init()

{

Container c=getContentPane();

js=new JScrollPane (l,ScrollPaneConstants.VERTICAL\_SCROLLBAR\_ALWAYS,ScrollPaneConstants.HORIZONTAL\_SCROLLBAR\_AS\_NEEDED);

add(js);

}

}

**ProgressBar**

Progress bar is a A component that visually displays the progress of some task. As the task progresses towards completion, the progress bar displays the task's percentage of completion. This percentage is typically represented visually by a rectangle which starts out empty and gradually becomes filled in as the task progresses. In addition, the progress bar can display a textual representation of this percentage.

Java provides JProgressBar class to implement progress bar.Its super class is JComponent.

|  |
| --- |
| **Constructor and Description** |
| [**JProgressBar**](https://docs.oracle.com/javase/7/docs/api/javax/swing/JProgressBar.html#JProgressBar())()  Creates a horizontal progress bar that displays a border but no progress string. |
|  |
| [**JProgressBar**](https://docs.oracle.com/javase/7/docs/api/javax/swing/JProgressBar.html#JProgressBar(int))(int orient)  Creates a progress bar with the specified orientation, which can be either SwingConstants.VERTICAL or SwingConstants.HORIZONTAL. |
| [**JProgressBar**](https://docs.oracle.com/javase/7/docs/api/javax/swing/JProgressBar.html#JProgressBar(int,%20int))(int min, int max)  Creates a horizontal progress bar with the specified minimum and maximum. |
| [**JProgressBar**](https://docs.oracle.com/javase/7/docs/api/javax/swing/JProgressBar.html#JProgressBar(int,%20int,%20int))(int orient, int min, int max)  Creates a progress bar using the specified orientation, minimum, and maximum. |

**Methods of JProgressBar**

1. int getMaximum():- Returns the progress bar's maximum value
2. int getMinimun():- Returns the progress bar's minimum value
3. int getOrientation(): Returns SwingConstants.VERTICAL or SwingConstants.HORIZONTAL,

depending on the orientation of the progress bar.

1. double getPercentComplete():-Returns the percent complete for the progress bar.
2. String getString():-Returns a String representation of the current progress.
3. int getValue():-Returns the progress bar's current value.
4. void setMaximum():-Sets the progress bar's maximum value to n.
5. void setMinimum():-Sets the progress bar's minimum value to n.
6. void setOrientation(int orien):- Sets the progress bar's orientation to newOrientation, which must be SwingConstants.VERTICAL orSwingConstants.HORIZONTAL.
7. void setString(String s):- Sets the value of the progress string.
8. Void setValue(int n):- Sets the progress bars current value to n.

**Example:**

import java.awt.\*;

import javax.swing.\*;

import java.awt.event.\*;

/\*<applet code=jprogress width=400 height=400>

</applet>\*/

public class jprogress extends JApplet implements ActionListener

{

JProgressBar jb=new JProgressBar();

JButton b=new JButton("Increment");

public void init()

{

Container c=getContentPane();

c.setLayout(new FlowLayout());

jb.setMinimum(0);

jb.setMaximum(200);

jb.setOrientation(JProgressBar.VERTICAL);

jb.setValue(50);

jb.setStringPainted(true);

c.add(jb);

c.add(b);

b.addActionListener(this);

}

public void actionPerformed(ActionEvent e)

{

jb.setValue(jb.getValue()+10);

}

}

**Tool Tips:**

This control is used to display a "Tip" for a Component. Typically components provide api to automate the process of using ToolTips. For example, any Swing component can use theJComponent setToolTipText method to specify the text for a standard tooltip. A component that wants to create a custom ToolTip display can override JComponent'screateToolTip method and use a subclass of this class.

JToolTip Is the class to create tool tips.

**Example:**

1. import javax.swing.JButton;

import javax.swing.JFrame;

public class ToolTipDemo {

public static void main(String args[]) {

JFrame frame = new JFrame();

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

JButton b = new JButton("Hello, World");

frame.add(b,"Center");

b.setToolTipText("This is a hello button");

frame.setSize(300, 200);

frame.setVisible(true);

}

}

1. import java.awt.Color;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JToolTip;

public class ToolTipCreation {

public static void main(String args[]) {

JFrame frame = new JFrame();

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

JButton b = new JButton("Hello, World") {

public JToolTip createToolTip() {

JToolTip tip = super.createToolTip();

tip.setBackground(Color.YELLOW);

tip.setForeground(Color.RED);

return tip;

}

};

frame.add(b,"Center");

b.setToolTipText("Hello button");

frame.setSize(300, 200);

frame.setVisible(true);

}

}

**Tables**

A *table* 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.

Tables are implemented by the **JTable** class, which extends **JComponent**. With the [JTable](http://download.oracle.com/javase/7/docs/api/javax/swing/JTable.html" \t "_blank) class you can display tables of data, optionally allowing the user to edit the data. JTabledoes not contain or cache data; it is simply a view of your data. Here is a picture of a typical table displayed within a scroll pane:

One of its constructors is shown here:

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.

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.

**Example**

import java.awt.\*;

import javax.swing.\*;

import javax.swing.table.\*;

public class tabl extends JFrame

{

JTextField f=new JTextField(20);

public tabl()

{

setSize(300,200);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

Object r[][]={{"abc",new Integer(10),"solapur"},{"pqr",new Integer(11),"Akkt"}};

String c[]={"Name","Roll call","Address"};

DefaultTableModel dtm=new DefaultTableModel(r,c);

JTable t=new JTable(dtm){

public boolean isCellEditable(int r,int c)

{

return false;//disallow editing of any cell

}

};

t.createDefaultColumnsFromModel();

JTableHeader h=t.getTableHeader();

h.setBackground(Color.yellow);

t.setBackground(Color.cyan);

t.setColumnSelectionAllowed(true);

t.setRowSelectionAllowed(true);

dtm.addColumn("Marks");

dtm.insertRow(1,new Object[]{"aaa",new Integer(12),"latur",new Float(70.0f)});

dtm.setValueAt("80.0",0,3);

dtm.setValueAt("81.0",2,3);

t.setGridColor(Color.red);

JScrollPane js=new JScrollPane(t);

getContentPane().add(f,BorderLayout.SOUTH);

getContentPane().add(js,BorderLayout.CENTER);

f.setText(t.getRowCount()+"rows and"+t.getColumnCount()+"columns");

}

public static void main(String ar[])

{

tabl l=new tabl();

l.setVisible(true);

}

}

**Tree**

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. A sampling of its constructors is shown here:

JTree(Object obj[ ])

JTree(Vector v)

JTree(TreeNode tn)

In the first form, the tree is constructed from the elements in the array obj. The second form constructs the tree from the elements of vector v. In the third form, the tree whose root node is specified by tn specifies the tree.

Although JTree is packaged in javax.swing, its support classes and interfaces are packaged in **javax.swing.tree**. This is because the number of classes and interfaces needed to support JTree is quite large.

JTree relies on two models: TreeModel and TreeSelectionModel.

A JTree generates a variety of events, but three relate specifically to trees: TreeExpansionEvent, TreeSelectionEvent, and TreeModelEvent.

TreeExpansionEvent events occur when a node is expanded or collapsed. A TreeSelectionEvent is generated when the user selects or deselects a node within the tree. A TreeModelEvent is fired when the data or structure of the tree changes. The listeners for these events are TreeExpansionListener, TreeSelectionListener, and TreeModelListener, respectively.

The tree event classes and listener interfaces are packaged **in javax.swing.event.**

When the TreeSelectionEvent is generated,to listen for this event, implement TreeSelectionListener. It defines only one method, called valueChanged( ), which receives the TreeSelectionEvent object.

You can obtain the path to the selected object by calling getPath( ), shown here, on the event object.

TreePath getPath( )

It returns a TreePath object that describes the path to the changed node.

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.

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.

**Example**

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

import javax.swing.tree.\*;

import javax.swing.event.\*;

//<applet code=jtree width=600 height=600></applet>

public class jtree extends JApplet implements TreeExpansionListener

{

Container c;

JTree tr;

DefaultMutableTreeNode top=new DefaultMutableTreeNode("Root");

DefaultMutableTreeNode first=new DefaultMutableTreeNode("1");

DefaultMutableTreeNode sec=new DefaultMutableTreeNode("2");

DefaultMutableTreeNode first\_1=new DefaultMutableTreeNode("1.1");

DefaultMutableTreeNode first\_2=new DefaultMutableTreeNode("1.2");

DefaultMutableTreeNode sec\_1=new DefaultMutableTreeNode("2.1");

DefaultMutableTreeNode sec\_2=new DefaultMutableTreeNode("2.2");

public void init()

{

c=getContentPane();

top.add(first);

top.add(sec);

first.add(first\_1);

first.add(first\_2);

sec.add(sec\_1);

sec.add(sec\_2);

tr=new JTree(top);

tr.addTreeExpansionListener(this);

c.add(tr);

}

public void treeExpanded(TreeExpansionEvent e)

{

TreePath tp=e.getPath();

showStatus("tree expanded at"+tp.toString());

}

public void treeCollapsed(TreeExpansionEvent e)

{

showStatus("tree collapsed");

}

}

**ToggleButton**

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. Toggle buttons are objects of the JToggleButton class.

JToggleButton implements AbstractButton. 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 two-state components.

JToggleButton defines several constructors. 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.

**Example:**

import java.awt.\*;

import javax.swing.\*;

/\*<applet code="toggle" width=400 height=400>

</applet>\*/

public class toggle extends JApplet

{

JToggleButton t=new JToggleButton("ok");

public void init()

{

Container c=getContentPane();

c.add(t);

//t.setPressedIcon(new ImageIcon("Tulips.jpg"));

}

}

**Separators**

Separators are horizontal or vertical bars that let you organize your controls into groups.Although mostly used in menus to divide options into logical groupings, separators can also be used in JApplet and Jframe components like any other components.

Separators are supported by JSeparator class in swing which is the subclass of JComponent.

Its constructors are as follows:

JSeparator()- It creates a new horizontal separator.

JSeparator(int orientation)- It creates a new separator with the indicated horizontal or vertical orientation.

Methods:

int getOrientation()- Returns the orientation of separator.

void setOrientation(int orientation)- Sets the new orientation for the separator.

When we add separator to contentpane it is not visible by deafult.To make it visible we need to set its preffered size using

void setPrefferedSize(Dimension d) method.

**Example:**

import java.awt.\*;

import javax.swing.\*;

/\*<applet code=separator width=400 height=400>

</applet>\*/

public class separator extends JApplet

{

JPanel p1,p2;

JButton b1,b2,b3,b4;

JSeparator s=new JSeparator(JSeparator.VERTICAL);

public void init()

{

Container c=getContentPane();

p1=new JPanel();

p2=new JPanel();

b1=new JButton("1");

b2=new JButton("2");

b3=new JButton("3");

b4=new JButton("4");

p1.add(b1);

p1.add(b2);

p2.add(b3);

p2.add(b4);

s.setPreferredSize(new Dimension(20,400));

c.setLayout(new FlowLayout());

c.add(p1);

c.add(s);

c.add(p2);

}

}