
This will cover java GUI programming concepts + chapters of “Advanced Java Programming” –Unit: 2 (BSc. CSIT, TU)

Swing and MVC Design Patterns

Swing is a part of **Java Foundation classes (JFC)**, the other parts of JFC are **java2D** and **Abstract window toolkit (AWT)**. AWT, Swing & Java 2D are used for building graphical user interfaces (GUIs) in java.

- Although the **AWT** is still a crucial part of Java, its component set is no longer widely used to create graphical user interfaces. Today, most programmers use **Swing** or **JavaFX** for this purpose.
- Swing is a framework that provides more powerful and flexible GUI components than does the AWT. As a result, it is the GUI that has been widely used by Java programmers for more than a decade

The Origins of Swing

- Swing did not exist in the early days of Java. Rather, it was a response to deficiencies present in Java’s original GUI subsystem: the Abstract Window Toolkit.
- The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface. One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding, platform-specific equivalents, or peers. This means that *the look and feel of a component is defined by the platform, not by Java*. Because the AWT components use native code resources, they are referred to as *heavyweight*.

Not long after Java’s original release, it became apparent that the limitations and restrictions present in the AWT were sufficiently serious that a better approach was needed. The solution was **Swing**. Introduced in 1997, Swing was included as part of the **Java Foundation Classes (JFC)**. Swing was initially available for use with Java 1.1 as a separate library. However, beginning with Java 1.2, Swing (and the rest of the JFC) was fully integrated into Java.

Swing Is Built on the AWT

Although Swing eliminates a number of the limitations inherent in the AWT, Swing does not replace it. Instead, Swing is built on the foundation of the AWT. This is why the AWT is still a crucial part of Java. Swing also uses the same event handling mechanism as the AWT. Therefore, a basic understanding of the AWT and of event handling is required to use Swing.

The MVC Connection

In general, a visual component is a composite of three distinct aspects:

- The way that the component looks when rendered on the screen
- The way that the component reacts to the user
- The state information associated with the component

No matter what architecture is used to implement a component, it must implicitly contain these three parts. Over the years, one component architecture has proven itself to be exceptionally effective: **Model-View-Controller**, or **MVC** for short.

The MVC architecture is successful because each piece of the design corresponds to an aspect of a component. In MVC terminology,

- The **model** corresponds to the state information associated with the component. For example, in the case of a check box, the model contains a field that indicates if the box is checked or unchecked.
- The **view** determines how the component is displayed on the screen, including any aspects of the view that are affected by the current state of the model.
- The **controller** determines how the component reacts to the user. For example, when the user clicks a check box, the controller reacts by changing the model to reflect the user's choice (checked or unchecked). This then results in the view being updated.

By separating a component into a **model**, a **view**, and a **controller**, the specific implementation of each can be changed without affecting the other two. For instance, different view implementations can render the same component in different ways without affecting the model or the controller.

Although the MVC architecture and the principles behind it are conceptually sound, the high level of separation between the view and the controller is not beneficial for Swing components. Instead, Swing uses a modified version of MVC that combines the view and the controller into a single logical entity called the **UI delegate**. For this reason, Swing's approach is called either the **Model-Delegate architecture** or the **Separable Model architecture**. Therefore, although Swing's component architecture is based on MVC, it does not use a classical implementation of it.

Swing's pluggable look and feel is made possible by its Model-Delegate architecture. Because the view (look) and controller (feel) are separate from the model, the look and feel can be changed without affecting how the component is used within a program. Conversely, it is possible to customize the model without affecting the way that the component appears on the screen or responds to user input.

To support the Model-Delegate architecture, most Swing components contain two objects. The first represents the model. The second represents the UI delegate. Models are defined by interfaces. For example, the model for a button is defined by the **ButtonModel** interface. UI delegates are classes that inherit **ComponentUI**. For example, the UI delegate for a button is **ButtonUI**. Normally, your programs will not interact directly with the UI delegate.

In summary,

- *Model* represents component's data.

- *View* represents visual representation of the component's data.
- *Controller* takes the input from the user on the view and reflects the changes in Component's data.
- Swing component has **Model** as a separate element, while the **View** and **Controller** part are clubbed in the User Interface elements. Because of which, Swing has a pluggable look-and-feel architecture.

AWT vs Swing

No.	Java AWT	Java Swing
1)	AWT components are platform-dependent.	Java swing components are platform-independent.
2)	AWT components are heavyweight.	Swing components are lightweight.
3)	AWT doesn't support pluggable look and feel.	Swing supports pluggable look and feel.
4)	AWT provides less components than Swing.	Swing provides more powerful components such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT doesn't follows MVC(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing follows MVC.

Swing Features

- **Light Weight** – Swing components are independent of native Operating System's API as Swing API controls are rendered mostly using pure JAVA code instead of underlying operating system calls.
- **Rich Controls** – Swing provides a rich set of advanced controls like Tree, TabbedPane, slider, colorpicker, and table controls.
- **Highly Customizable** – Swing controls can be customized in a very easy way as visual appearance is independent of internal representation.
- **Pluggable look-and-feel** – Swing based GUI Application look and feel can be changed at run-time, based on available values.

Swing API is a set of extensible GUI Components to ease the developer's life to create JAVA based Front End/GUI Applications. It is built on top of AWT API and acts as a replacement of AWT API, since it has almost every control corresponding to AWT controls. Swing component follows a Model-View-Controller architecture to fulfill the following criterias.

- A single API is to be sufficient to support multiple look and feel.
- API is to be model driven so that the highest level API is not required to have data.
- API is to use the Java Bean model so that Builder Tools and IDE can provide better services to the developers for use.

Hierarchy of Java Swing classes

The hierarchy of java swing API is given below.

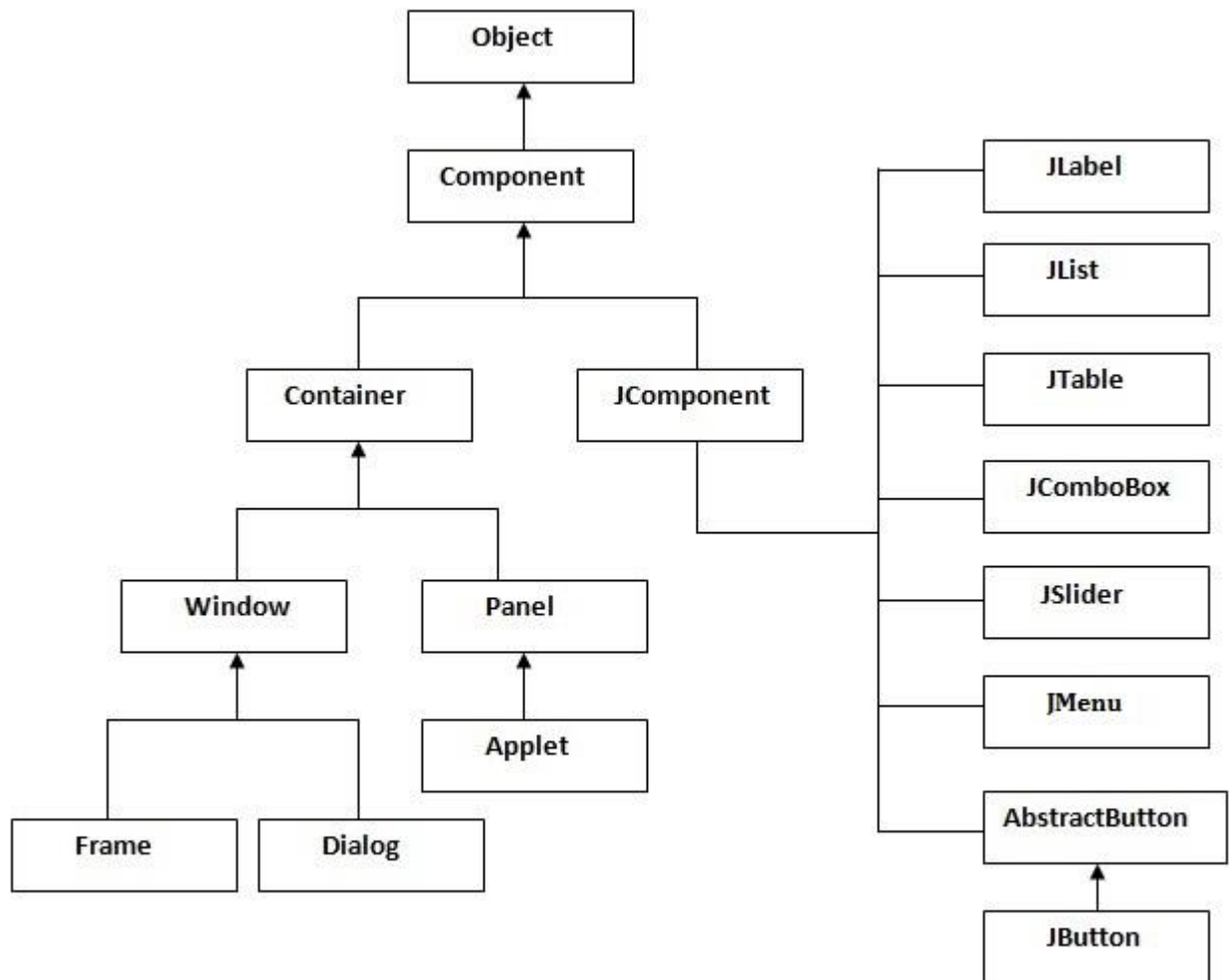


Figure 1: Hierarchy of Java Swing Classes

SWING Containers

Containers are an integral part of SWING GUI components. A container provides a space where a component can be located. A Container in AWT is a component itself and it provides the capability to add a component to itself. Following are certain noticable points to be considered.

- Sub classes of Container are called as Container. For example, **JPanel**, **JFrame** and **JWindow**.
- Container can add only a Component to itself.
- A default layout is present in each container which can be overridden using `setLayout` method.

Following is the list of commonly used containers while designed GUI using SWING.

Sr.No.	Container & Description
1	Panel <i>JPanel</i> is the simplest container. It provides space in which any other component can be placed, including other panels.
2	Frame A <i>JFrame</i> is a top-level window with a title and a border.
3	Window A <i>JWindow</i> object is a top-level window with no borders and no menubar.

- To appear onscreen, every GUI component must be part of a containment hierarchy. A containment hierarchy is a tree of components that has a top-level container as its root. We'll show you one in a bit.
- Each GUI component can be contained only once. If a component is already in a container and you try to add it to another container, the component will be removed from the first container and then added to the second.
- Each top-level container has a content pane that, generally speaking, contains (directly or indirectly) the visible components in that top-level container's GUI.
- You can optionally add a menu bar to a top-level container. The menu bar is by convention positioned within the top-level container, but outside the content pane. Some look and feels, such as the Mac OS look and feel, give you the option of placing the menu bar in another place more appropriate for the look and feel, such as at the top of the screen.

- ❖ Swing defines two types of containers: **heavy weight containers** and **light weight containers**
- ❖ **Heavy weight containers** are top-level containers and are not inherited from **JComponent** rather they are directly inherited from AWT classes **Component** and **Containers**
- ❖ A top level container is not contained within any other containers.
- ❖ Every containment hierarchy must begin with a top-level container.
- ❖ JWindow, JFrame, JDialog, and, JApplet are useful top level containers provided by Swing.
- ❖ **Light weight containers** are inherited from **JComponent**.
- ❖ **JPanel** is an example of Light weight container.
- ❖ Light weight containers are often used to organize and manage groups of related components.
- ❖ Light weight containers can be contained within another container.
- ❖ Each top level containers contains the following panes:
 - **Root Pane:** intermediate container that manages the other panes. It can also manage an optional menu bar.
 - **Glass Pane:** Hidden, by default. If you make the glass pane visible, then it's like a sheet of glass over all the other parts of the root pane. It's completely transparent unless you implement the glass pane's *paintComponent* method so that it does something, and it can intercept input events for the root pane.
 - **Layered Pane:** area where most of the actions take place. Menus and dialogue boxes are opened in this pane.
 - **Content Pane:** holds all visible components of the root pane, except the menu bar. It covers the visible section of the JFrame or Jwindow and we use it to add components to the display area.

SWING – Controls

Every user interface considers the following three main aspects –

- **UI Elements** – These are the core visual elements the user eventually sees and interacts with.
- **Layouts** – They define how UI elements should be organized on the screen and provide a final look and feel to the GUI (Graphical User Interface).
- **Behavior** – These are the events which occur when the user interacts with UI elements.

Layout Management

Layout refers to the arrangement of components within the container. In another way, it could be said that layout is placing the components at a particular position within the container. The task of laying out the controls is done automatically by the Layout Manager.

Layout Manager

The layout manager automatically positions all the components within the container. Even if you do not use the layout manager, the components are still positioned by the default layout manager. It is possible to lay out the controls by hand, however, it becomes very difficult because of the following two reasons.

- ✓ It is very tedious to handle a large number of controls within the container.
- ✓ Usually, the width and height information of a component is not given when we need to arrange them.

Java provides various layout managers to position the controls. Properties like size, shape, and arrangement varies from one layout manager to the other. When the size of the applet or the application window changes, the size, shape, and arrangement of the components also changes in response, i.e. the layout managers adapt to the dimensions of the appletviewer or the application window.

The layout manager is associated with every Container object. Each layout manager is an object of the class that implements the **LayoutManager** interface.

Following are the interfaces defining the functionalities of Layout Managers.

Sr.No.	Interface & Description
1	LayoutManager The LayoutManager interface declares those methods which need to be implemented by the class, whose object will act as a layout manager.
2	LayoutManager2

	<p>The <code>LayoutManager2</code> is the sub-interface of the <code>LayoutManager</code>. This interface is for those classes that know how to layout containers based on layout constraint object.</p>
--	--

AWT Layout Manager Classes

Following is the list of commonly used controls while designing GUI using AWT.

Sr.No.	LayoutManager & Description
1	BorderLayout The BorderLayout arranges the components to fit in the five regions: east, west, north, south, and center.
2	CardLayout The CardLayout object treats each component in the container as a card. Only one card is visible at a time.
3	FlowLayout The FlowLayout is the default layout. It layout the components in a directional flow.
4	GridLayout The GridLayout manages the components in the form of a rectangular grid.
5	GridBagLayout This is the most flexible layout manager class. The object of GridBagLayout aligns the component vertically, horizontally, or along their baseline without requiring the components of the same size.
6	GroupLayout The GroupLayout hierarchically groups the components in order to position them in a Container.
7	SpringLayout A SpringLayout positions the children of its associated container according to a set of constraints.

There are following classes that represents the layout managers:

- java.awt.BorderLayout
- java.awt.FlowLayout
- java.awt.GridLayout
- java.awt.CardLayout
- java.awt.GridBagLayout
- javax.swing.BoxLayout
- javax.swing.GroupLayout
- javax.swing.ScrollPaneLayout
- javax.swing.SpringLayout etc.

BorderLayout

The BorderLayout is used to arrange the components in five regions: north, south, east, west and center. Each region (area) may contain one component only. It is the default layout of frame or window. The BorderLayout provides five constants for each region.

Constructors of BorderLayout class:

- ✓ **BorderLayout():** creates a border layout but with no gaps between the components.
- ✓ **JBorderLayout(int hgap, int vgap):** creates a border layout with the given horizontal and vertical gaps between the components.

Example of BorderLayout

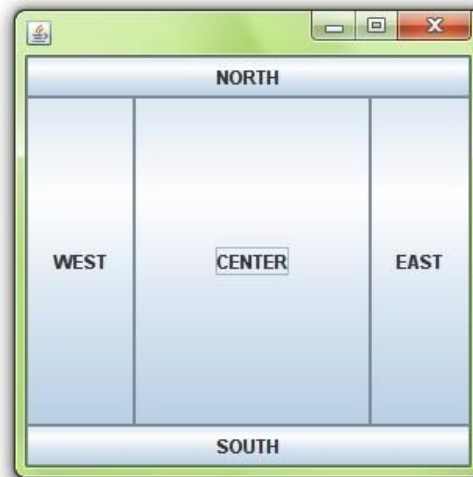


Figure 2: BorderLayout

1. **import** java.awt.*;
2. **import** javax.swing.*;
- 3.
4. **public class** Border {
5. JFrame f;

```

6. Border(){
7.     f=new JFrame();
8.
9.     JButton b1=new JButton("NORTH");
10.    JButton b2=new JButton("SOUTH");
11.    JButton b3=new JButton("EAST");
12.    JButton b4=new JButton("WEST");
13.    JButton b5=new JButton("CENTER");
14.
15.    f.add(b1,BorderLayout.NORTH);
16.    f.add(b2,BorderLayout.SOUTH);
17.    f.add(b3,BorderLayout.EAST);
18.    f.add(b4,BorderLayout.WEST);
19.    f.add(b5,BorderLayout.CENTER);
20.
21.    f.setSize(300,300);
22.    f.setVisible(true);
23. }
24. public static void main(String[] args) {
25.     new Border();
26. }
27. }

```

GridLayout

The GridLayout is used to arrange the components in rectangular grid. One component is displayed in each rectangle.

Constructors of GridLayout class

- **GridLayout():** creates a grid layout with one column per component in a row.
- **GridLayout(int rows, int columns):** creates a grid layout with the given rows and columns but no gaps between the components.
- **GridLayout(int rows, int columns, int hgap, int vgap):** creates a grid layout with the given rows and columns alongwith given horizontal and vertical gaps.

Example of GridLayout class



Figure 3:GridLayout

```

1. import java.awt.*;
2. import javax.swing.*;
3.
4. public class MyGridLayout{
5. JFrame f;
6. MyGridLayout(){
7.     f=new JFrame();
8.
9.     JButton b1=new JButton("1");
10.    JButton b2=new JButton("2");
11.    JButton b3=new JButton("3");
12.    JButton b4=new JButton("4");
13.    JButton b5=new JButton("5");
14.        JButton b6=new JButton("6");
15.        JButton b7=new JButton("7");
16.    JButton b8=new JButton("8");
17.        JButton b9=new JButton("9");
18.
19.    f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);
20.    f.add(b6);f.add(b7);f.add(b8);f.add(b9);
21.
22.    f.setLayout(new GridLayout(3,3));
23.    //setting grid layout of 3 rows and 3 columns
24.
25.    f.setSize(300,300);
26.    f.setVisible(true);
27. }
28. public static void main(String[] args) {
29.     new MyGridLayout();
30. }
31. }

```

GridBagLayout

The Java GridBagLayout class is used to align components vertically, horizontally or along their baseline.

The components may not be of same size. Each GridBagLayout object maintains a dynamic, rectangular grid of cells. Each component occupies one or more cells known as its display area. Each component associates an instance of GridBagConstraints. With the help of constraints object we arrange component's display area on the grid. The GridBagLayout manages each component's minimum and preferred sizes in order to determine component's size.

Example of GridBagLayout

```

1. import java.awt.Button;
2. import java.awt.GridBagConstraints;
3. import java.awt.GridBagLayout;
4.
5. import javax.swing.*;
6. public class GridBagLayoutExample extends JFrame{
7.     public static void main(String[] args) {
8.         GridBagLayoutExample a = new GridBagLayoutExample();
9.     }
10.    public GridBagLayoutExample() {
11.        GridBagLayoutgrid = new GridBagLayout();
12.        GridBagConstraints gbc = new GridBagConstraints();
13.        setLayout(grid);
14.        setTitle("GridBag Layout Example");
15.        GridBagLayout layout = new GridBagLayout();
16.        this.setLayout(layout);
17.        gbc.fill = GridBagConstraints.HORIZONTAL;
18.        gbc.gridx = 0;
19.        gbc.gridy = 0;
20.        this.add(new Button("Button One"), gbc);
21.        gbc.gridx = 1;
22.        gbc.gridy = 0;
23.        this.add(new Button("Button two"), gbc);
24.        gbc.fill = GridBagConstraints.HORIZONTAL;
25.        gbc.ipady = 20;
26.        gbc.gridx = 0;
27.        gbc.gridy = 1;
28.        this.add(new Button("Button Three"), gbc);
29.        gbc.gridx = 1;
30.        gbc.gridy = 1;
31.        this.add(new Button("Button Four"), gbc);
32.        gbc.gridx = 0;
33.        gbc.gridy = 2;
34.        gbc.fill = GridBagConstraints.HORIZONTAL;
35.        gbc.gridwidth = 2;
36.        this.add(new Button("Button Five"), gbc);
37.        setSize(300, 300);
38.        setPreferredSize(getSize());

```

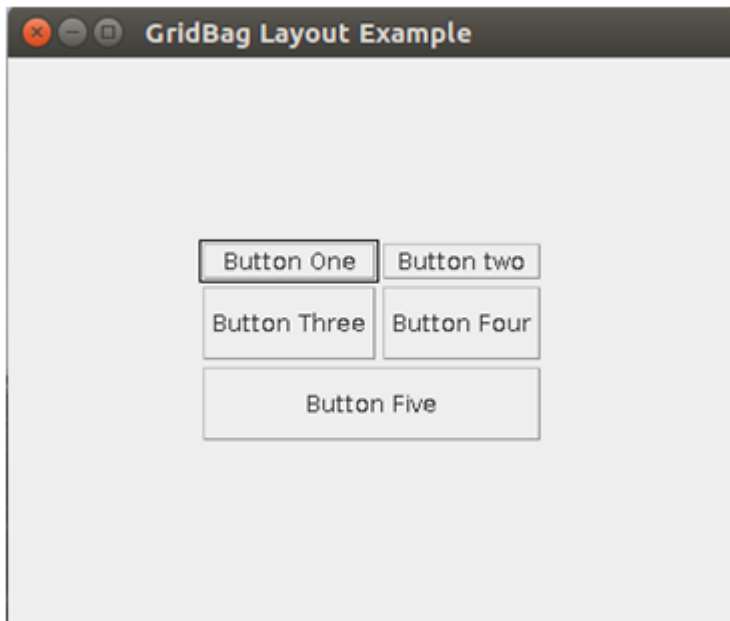


```

39.         setVisible(true);
40.         setDefaultCloseOperation(EXIT_ON_CLOSE);
41.
42.     }
43.
44. }

```

Output:



Fields in GridBagLayout

Modifier and Type	Field	Description
double[]	columnWeights	It is used to hold the overrides to the column weights.
int[]	columnWidths	It is used to hold the overrides to the column minimum width.

protected Hashtable<Component,GridBagConstraints>	comptable	It is used to maintains the association between a component and its gridbag constraints.
protected GridBagConstraints	defaultConstraints	It is used to hold a gridbag constraints instance containing the default values.
protected GridBagLayoutInfo	layoutInfo	It is used to hold the layout information for the gridbag.
protected static int	MAXGRIDSIZE	No longer in use just for backward compatibility
protected static int	MINSIZE	It is smallest grid that can be laid out by the grid bag layout.
protected static int	PREFERRED_SIZE	It is preferred grid size that can be laid out by the grid bag layout.
int[]	rowHeights	It is used to hold the overrides to the row minimum heights.
double[]	rowWeights	It is used to hold the overrides to the row weights

Useful Methods

Modifier and Type	Method	Description
void	addLayoutComponent(Component comp, Object constraints)	It adds specified component to the layout, using the specified constraints object.
void	addLayoutComponent(String name, Component comp)	It has no effect, since this layout manager does not use a per-component string.
protected void	adjustForGravity(GridBagConstraints constraints, Rectangle r)	It adjusts the x, y, width, and height fields to the correct values depending on the constraint geometry and pads.
protected void	AdjustForGravity(GridBagConstraints constraints, Rectangle r)	This method is for backwards compatibility only
protected void	arrangeGrid(Container parent)	Lays out the grid.
protected void	ArrangeGrid(Container parent)	This method is obsolete and supplied for backwards compatibility
GridBagConstraints	getConstraints(Component comp)	It is for getting the constraints for the specified component.

float	getLayoutAlignmentX(Container parent)	It returns the alignment along the x axis.
float	getLayoutAlignmentY(Container parent)	It returns the alignment along the y axis.
int[][]	getLayoutDimensions()	It determines column widths and row heights for the layout grid.
protected GridBagLayoutInfo	getLayoutInfo(Container parent, int sizeflag)	This method is obsolete and supplied for backwards compatibility.
protected GridBagLayoutInfo	GetLayoutInfo(Container parent, int sizeflag)	This method is obsolete and supplied for backwards compatibility.
Point	getLayoutOrigin()	It determines the origin of the layout area, in the graphics coordinate space of the target container.
double[][]	getLayoutWeights()	It determines the weights of the layout grid's columns and rows.
protected Dimension	getMinSize(Container parent, GridBagLayoutInfo info)	It figures out the minimum size of the master based on the information from getLayoutInfo.
protected Dimension	GetMinSize(Container parent, GridBagLayoutInfo info)	This method is obsolete and supplied for backwards compatibility only

GroupLayout

GroupLayout groups its components and places them in a Container hierarchically. The grouping is done by instances of the Group class.

Group is an abstract class and two concrete classes which implement this Group class are SequentialGroup and ParallelGroup.

SequentialGroup positions its child sequentially one after another where as ParallelGroup aligns its child on top of each other.

The GroupLayout class provides methods such as createParallelGroup() and createSequentialGroup() to create groups.

GroupLayout treats each axis independently. That is, there is a group representing the horizontal axis, and a group representing the vertical axis. Each component must exist in both a horizontal and vertical group, otherwise an IllegalStateException is thrown during layout, or when the minimum, preferred or maximum size is requested.

Fields

Modifier and Type	Field	Description
static int	DEFAULT_SIZE	It indicates the size from the component or gap should be used for a particular range value.
static int	PREFERRED_SIZE	It indicates the preferred size from the component or gap should be used for a particular range value.

Constructors

GroupLayout(Container host)	It creates a GroupLayout for the specified Container.
-----------------------------	---

Nested Classes

Modifier and Type	Class	Description
static class	GroupLayout.Alignment	Enumeration of the possible ways GroupLayout can align its children.
class	GroupLayout.Group	Group provides the basis for the two types of operations supported by GroupLayout: laying out components one after another (SequentialGroup) or aligned (ParallelGroup).
class	GroupLayout.ParallelGroup	It is a Group that aligns and sizes it's children.
class	GroupLayout.SequentialGroup	It is a Group that positions and sizes its elements sequentially, one after another.

Useful Methods

Modifier and Type	Field	Description
void	addLayoutComponent(Component component, Object constraints)	It notify that a Component has been added to the parent container.
void	addLayoutComponent(String name, Component component)	It notify that a Component has been added to the parent container.
GroupLayout.ParallelGroup	createBaselineGroup(boolean resizable, boolean anchorBaselineToTop)	It creates and returns a ParallelGroup that

		aligns it's elements along the baseline.
GroupLayout.ParallelGroup	createParallelGroup()	It creates and returns a GroupLayout with an alignment of Alignment.LEADING
GroupLayout.ParallelGroup	createParallelGroup(GroupLayout.Alignment alignment)	It creates and returns a GroupLayout with the specified alignment.
GroupLayout.ParallelGroup	createParallelGroup(GroupLayout.Alignment alignment, boolean resizable)	It creates and returns a GroupLayout with the specified alignment and resize behavior.
GroupLayout.SequentialGroup	createSequentialGroup()	It creates and returns aSequentialGroup.
boolean	getAutoCreateContainerGaps()	It returns true if gaps between the container and components that border the container are automatically created.
boolean	getAutoCreateGaps()	It returns true if gaps between components are automatically created.

boolean	getHonorsVisibility()	It returns whether component visibility is considered when sizing and positioning components.
float	getLayoutAlignmentX(Container parent)	It returns the alignment along the x axis.
float	getLayoutAlignmentY(Container parent)	It returns the alignment along the y axis.
Dimension	maximumLayoutSize(Container parent)	It returns the maximum size for the specified container.

Example

```

1. import java.awt.Container;
2. import javax.swing.GroupLayout;
3. import javax.swing.JButton;
4. import javax.swing.JFrame;
5. import static javax.swing.GroupLayout.Alignment.*;
6. public class GroupExample2 {
7.     public static void main(String[] args) {
8.         JFrame frame = new JFrame("GroupLayoutExample");
9.         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10.        Container myPanel = frame.getContentPane();
11.
12.        GroupLayout groupLayout = new GroupLayout(myPanel);
13.        groupLayout.setAutoCreateGaps(true);
14.        groupLayout.setAutoCreateContainerGaps(true);
15.        myPanel.setLayout(groupLayout);
16.
17.        JButton b1 = new JButton("Button One");
18.        JButton b2 = new JButton("Button Two");

```



```

19.    JButton b3 = new JButton("Button Three");
20.
21.    groupLayout.setHorizontalGroup(groupLayout.createSequentialGroup()
22.        .addGroup(groupLayout.createParallelGroup(LEADING).addComponent(b1).ad
23.            dComponent(b3))
24.        .addGroup(groupLayout.createParallelGroup(TRAILING).addComponent(b2)));
25.
26.    groupLayout.setVerticalGroup(groupLayout.createSequentialGroup()
27.        .addGroup(groupLayout.createParallelGroup(BASELINE).addComponent(b1).a
28.            ddComponent(b2))
29.        .addGroup(groupLayout.createParallelGroup(BASELINE).addComponent(b3)));
30.
31.    frame.pack();
32.    frame.setVisible(true);
33. }
34. }

```



Figure 4: GroupLayout

Using NO layout managers (Absolute Positioning)

Although it is possible to do without a layout manager, you should use a layout manager if at all possible. A layout manager makes it easier to adjust to look-and-feel-dependent component appearances, to different font sizes, to a container's changing size, and to different locales. Layout managers also can be reused easily by other containers, as well as other programs.

If a container holds components whose size is not affected by the container's size or by font, look-and-feel, or language changes, then absolute positioning might make sense. Desktop panes, which contain internal frames, are in this category. The size and position of internal frames does not depend directly on the desktop pane's size. The programmer determines the initial size and placement of internal frames within the desktop pane, and then the user can move or resize the frames. A layout manager is unnecessary in this situation.

Another situation in which absolute positioning might make sense is that of a custom container that performs size and position calculations that are particular to the container, and perhaps require knowledge of the container's specialized state. This is the situation with split panes.

Creating a container without a layout manager involves the following steps.

- Set the container's layout manager to null by calling *setLayout(null)*.
- Call the *Component* class's *setBounds* method for each of the container's children.
- Call the *Component* class's *repaint* method.

However, creating containers with absolutely positioned containers can cause problems if the window containing the container is resized.

Here is a snapshot of a frame whose content pane uses absolute positioning.

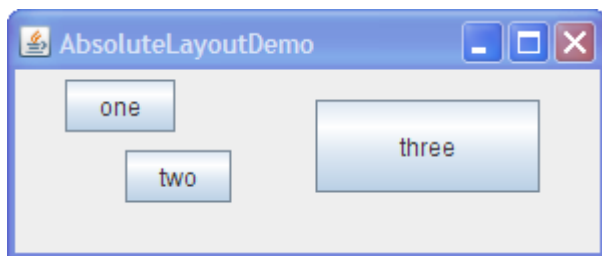


Figure 5: Using No LayoutManager

Example

```
pane.setLayout(null);

JButton b1 = new JButton("one");
JButton b2 = new JButton("two");
JButton b3 = new JButton("three");

pane.add(b1);
pane.add(b2);
pane.add(b3);

Insets insets = pane.getInsets();
Dimension size = b1.getPreferredSize();
b1.setBounds(25 + insets.left, 5 + insets.top,
             size.width, size.height);
size = b2.getPreferredSize();
b2.setBounds(55 + insets.left, 40 + insets.top,
             size.width, size.height);
size = b3.getPreferredSize();
b3.setBounds(150 + insets.left, 15 + insets.top,
             size.width + 50, size.height + 20);

...//In the main method:
Insets insets = frame.getInsets();
frame.setSize(300 + insets.left + insets.right,
             125 + insets.top + insets.bottom);
```

Custom Layout Managers

Before you start creating a custom layout manager, make sure that no existing layout manager meets your requirements. In particular, layout managers such as **GridBagLayout**, **SpringLayout**, and **BoxLayout** are flexible enough to work in many cases. You can also find layout managers from other sources, such as from the Internet. Finally, you can simplify layout by grouping components into containers such as panels

To create a custom layout manager, you must create a class that implements the **LayoutManager** interface. You can either implement it directly, or implement its subinterface, **LayoutManager2**.

Every layout manager must implement at least the following five methods, which are required by the **LayoutManager** interface:

void addLayoutComponent(String, Component)

Called by the Container class's add methods. Layout managers that do not associate strings with their components generally do nothing in this method.

void removeLayoutComponent(Component)

Called by the Container methods remove and removeAll. Layout managers override this method to clear an internal state they may have associated with the Component.

Dimension preferredLayoutSize(Container)

Called by the Container class's getPreferredSize method, which is itself called under a variety of circumstances. This method should calculate and return the ideal size of the container, assuming that the components it contains will be at or above their preferred sizes. This method must take into account the container's internal borders, which are returned by the getInsets method.

Dimension minimumLayoutSize(Container)

Called by the Container getMinimumSize method, which is itself called under a variety of circumstances. This method should calculate and return the minimum size of the container, assuming that the components it contains will be at or above their minimum sizes. This method must take into account the container's internal borders, which are returned by the getInsets method.

void layoutContainer(Container)

Called to position and size each of the components in the container. A layout manager's layoutContainer method does not actually draw components. It simply invokes one or more of each component's setSize, setLocation, and setBounds methods to set the component's size and position.

This method must take into account the container's internal borders, which are returned by the getInsets method. If appropriate, it should also take the container's orientation (returned by the getComponentOrientation method) into account. You cannot assume that

the `preferredLayoutSize` or `minimumLayoutSize` methods will be called before `layoutContainer` is called.

Besides implementing the preceding five methods, layout managers generally implement at least one public constructor and the **`toString`** method.

If you wish to support component constraints, maximum sizes, or alignment, then your layout manager should implement the `LayoutManager2` interface. In fact, for these reasons among many others, nearly all modern layout managers will need to implement `LayoutManager2`. That interface adds five methods to those required by `LayoutManager`:

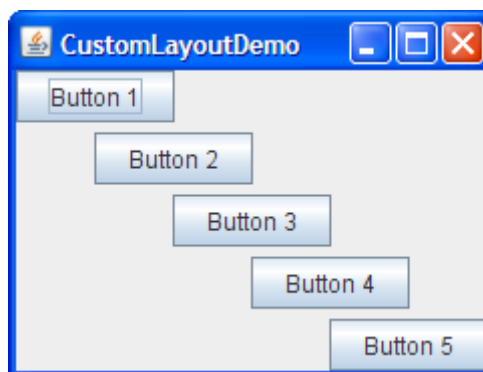
- `addLayoutComponent(Component, Object)`
- `getLayoutAlignmentX(Container)`
- `getLayoutAlignmentY(Container)`
- `invalidateLayout(Container)`
- `maximumLayoutSize(Container)`

Of these methods, the most important are `addLayoutComponent(Component, Object)` and `invalidateLayout(Container)`. The `addLayoutComponent` method is used to add components to the layout, using the specified constraint object. The `invalidateLayout` method is used to invalidate the layout, so that if the layout manager has cached information, this should be discarded.

Finally, whenever you create custom layout managers, you should be careful of keeping references to **`Component`** instances that are no longer children of the `Container`. Namely, layout managers should override **`removeLayoutComponent`** to clear any cached state related to the `Component`.

Example of a Custom Layout

The example `CustomLayoutDemo` uses a custom layout manager called `DiagonalLayout`. `DiagonalLayout` lays out components diagonally, from left to right, with one component per row. Here is a picture of `CustomLayoutDemo` using `DiagonalLayout` to lay out five buttons.



Code:

```

package layout;

/*
 * 1.2+ version.  Used by CustomLayoutDemo.java.
 */

import java.awt.*;

public class DiagonalLayout implements LayoutManager {
    private int vgap;
    private int minWidth = 0, minHeight = 0;
    private int preferredWidth = 0, preferredHeight = 0;
    private boolean sizeUnknown = true;

    public DiagonalLayout() {
        this(5);
    }

    public DiagonalLayout(int v) {
        vgap = v;
    }

    /* Required by LayoutManager. */
    public void addLayoutComponent(String name, Component comp) {
    }

    /* Required by LayoutManager. */
    public void removeLayoutComponent(Component comp) {
    }

    private void setSizes(Container parent) {
        int nComps = parent.getComponentCount();
        Dimension d = null;

        //Reset preferred/minimum width and height.
        preferredWidth = 0;
        preferredHeight = 0;
        minWidth = 0;
        minHeight = 0;

        for (int i = 0; i < nComps; i++) {
            Component c = parent.getComponent(i);
            if (c.isVisible()) {
                d = c.getPreferredSize();

                if (i > 0) {
                    preferredWidth += d.width/2;
                    preferredHeight += vgap;
                } else {
                    preferredWidth = d.width;
                }
                preferredHeight += d.height;

                minWidth = Math.max(c.getMinimumSize().width,
                                    minWidth);
                minHeight = preferredHeight;
            }
        }
    }
}

```

```

/* Required by LayoutManager. */
public Dimension preferredLayoutSize(Container parent) {
    Dimension dim = new Dimension(0, 0);
    int nComps = parent.getComponentCount();

    setSizes(parent);

    //Always add the container's insets!
    Insets insets = parent.getInsets();
    dim.width = preferredWidth
        + insets.left + insets.right;
    dim.height = preferredHeight
        + insets.top + insets.bottom;

    sizeUnknown = false;

    return dim;
}

/* Required by LayoutManager. */
public Dimension minimumLayoutSize(Container parent) {
    Dimension dim = new Dimension(0, 0);
    int nComps = parent.getComponentCount();

    //Always add the container's insets!
    Insets insets = parent.getInsets();
    dim.width = minWidth
        + insets.left + insets.right;
    dim.height = minHeight
        + insets.top + insets.bottom;

    sizeUnknown = false;

    return dim;
}

/* Required by LayoutManager. */
/*
 * This is called when the panel is first displayed,
 * and every time its size changes.
 * Note: You CAN'T assume preferredLayoutSize or
 * minimumLayoutSize will be called -- in the case
 * of applets, at least, they probably won't be.
 */
public void layoutContainer(Container parent) {
    Insets insets = parent.getInsets();
    int maxWidth = parent.getWidth()
        - (insets.left + insets.right);
    int maxHeight = parent.getHeight()
        - (insets.top + insets.bottom);
    int nComps = parent.getComponentCount();
    int previousWidth = 0, previousHeight = 0;
    int x = 0, y = insets.top;
    int rowh = 0, start = 0;
    int xFudge = 0, yFudge = 0;
    boolean oneColumn = false;

    // Go through the components' sizes, if neither
    // preferredLayoutSize nor minimumLayoutSize has

```

```

// been called.
if (sizeUnknown) {
    setSizes(parent);
}

if (maxWidth <= minWidth) {
    oneColumn = true;
}

if (maxWidth != preferredWidth) {
    xFudge = (maxWidth - preferredWidth)/(nComps - 1);
}

if (maxHeight > preferredHeight) {
    yFudge = (maxHeight - preferredHeight)/(nComps - 1);
}

for (int i = 0 ; i < nComps ; i++) {
    Component c = parent.getComponent(i);
    if (c.isVisible()) {
        Dimension d = c.getPreferredSize();

        // increase x and y, if appropriate
        if (i > 0) {
            if (!oneColumn) {
                x += previousWidth/2 + xFudge;
            }
            y += previousHeight + vgap + yFudge;
        }

        // If x is too large,
        if ((!oneColumn) &&
            (x + d.width) >
            (parent.getWidth() - insets.right)) {
            // reduce x to a reasonable number.
            x = parent.getWidth()
                - insets.bottom - d.width;
        }

        // If y is too large,
        if ((y + d.height)
            > (parent.getHeight() - insets.bottom)) {
            // do nothing.
            // Another choice would be to do what we do to x.
        }

        // Set the component's size and position.
        c.setBounds(x, y, d.width, d.height);

        previousWidth = d.width;
        previousHeight = d.height;
    }
}

public String toString() {
    String str = "";
    return getClass().getName() + "[vgap=" + vgap + str + "];"
}
}

```

Using Text Components

Swing provides six text components, along with supporting classes and interfaces that meet even the most complex text requirements. In spite of their different uses and capabilities, all Swing text components inherit from the same superclass, **JTextComponent**, which provides a highly-configurable and powerful foundation for text manipulation.

The following figure shows the **JTextComponent** hierarchy.

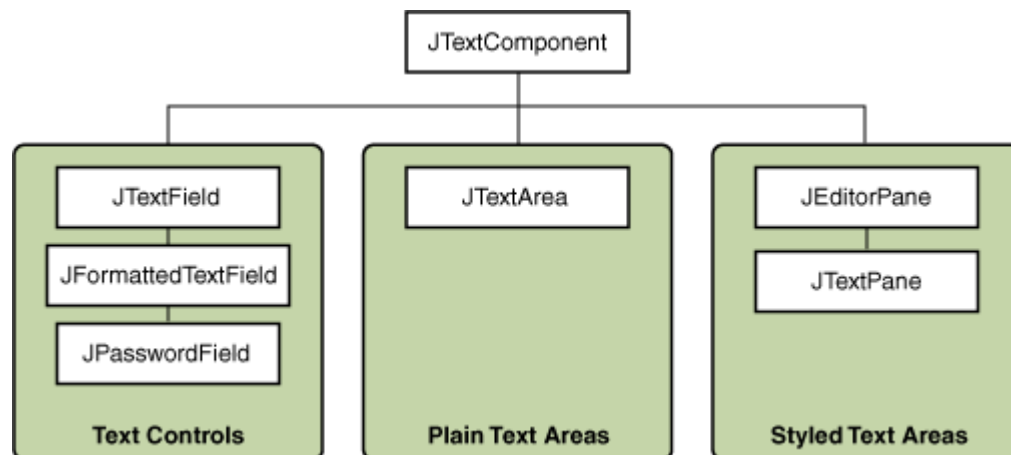


Figure 6: *JTextComponent* Hierarchy

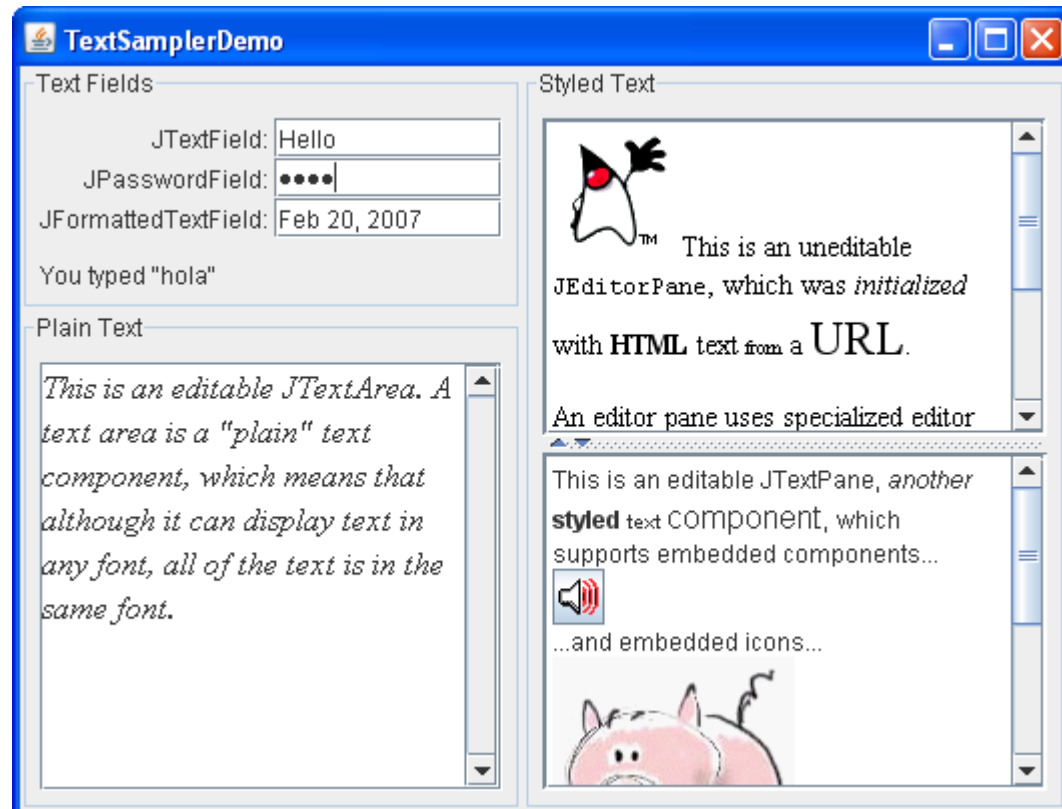


Figure 7: *TextDemo*

Java JTextField

The object of a JTextField class is a text component that allows the editing of a single line text. It inherits JTextComponent class.

JTextField class declaration

Declaration for javax.swing.JTextField class.

public class JTextField **extends** JTextComponent **implements** SwingConstants

Commonly used Constructors:

Constructor	Description
JTextField()	Creates a new TextField
JTextField(String text)	Creates a new TextField initialized with the specified text.
JTextField(String text, int columns)	Creates a new TextField initialized with the specified text and columns.
JTextField(int columns)	Creates a new empty TextField with the specified number of columns.

Commonly used Methods:

Methods	Description
void addActionListener(ActionListener l)	It is used to add the specified action listener to receive action events from this textfield.
Action getAction()	It returns the currently set Action for this ActionEvent source, or null if no Action is set.
void setFont(Font f)	It is used to set the current font.

<pre>void removeActionListener(ActionListener l)</pre>	<p>It is used to remove the specified action listener so that it no longer receives action events from this textfield.</p>
--	--

Example:

```
1. import javax.swing.*;
2. class TextFieldExample
3. {
4.     public static void main(String args[])
5.     {
6.         JFrame f= new JFrame("TextField Example");
7.         JTextField t1,t2;
8.         t1=new JTextField("TU-CSIT");
9.         t1.setBounds(50,100, 200,30);
10.        t2=new JTextField("Bipin Timalsina");
11.        t2.setBounds(50,150, 200,30);
12.        f.add(t1); f.add(t2);
13.        f.setSize(400,400);
14.        f.setLayout(null);
15.        f.setVisible(true);
16.    }
17. }
```

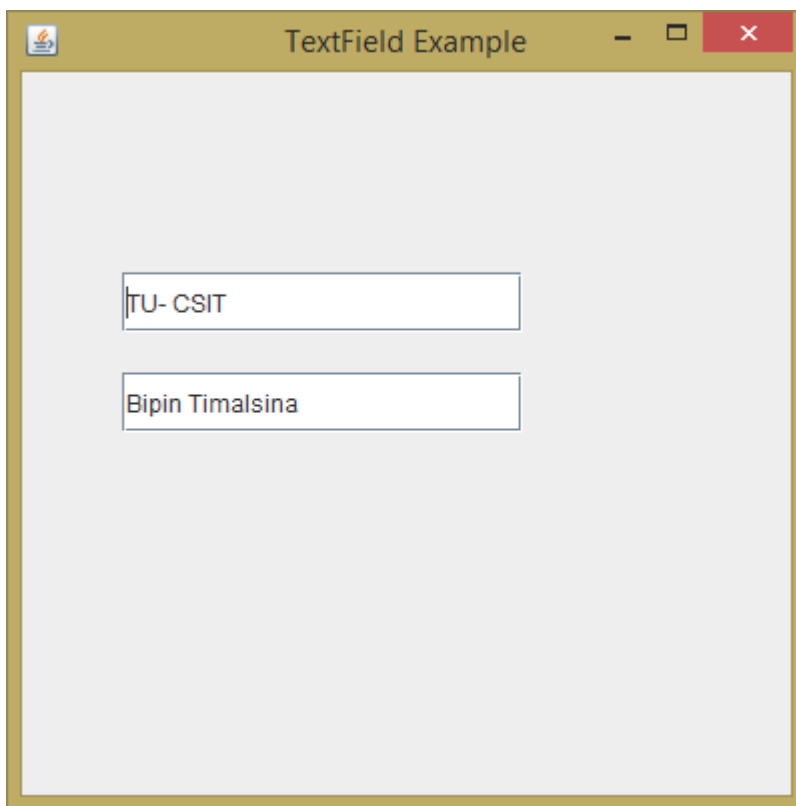


Figure 8: TextField Example

Java JPasswordField

The object of a **JPasswordField** class is a text component specialized for password entry. It allows the editing of a single line of text. It inherits JTextField class.

public class JPasswordField **extends** JTextField

Commonly used Constructors:

Constructor	Description
JPasswordField()	Constructs a new JPasswordField, with a default document, null starting text string, and 0 column width.
JPasswordField(int columns)	Constructs a new empty JPasswordField with the specified number of columns.
JPasswordField(String text)	Constructs a new JPasswordField initialized with the specified text.
JPasswordField(String text, int columns)	Construct a new JPasswordField initialized with the specified text and columns.

Example:

```

1. import javax.swing.*;
2. public class PasswordFieldExample {
3.     public static void main(String[] args) {
4.         JFrame f=new JFrame("Password Field Example");
5.         JPasswordField value = new JPasswordField();
6.         JLabel l1=new JLabel("Password:");
7.         l1.setBounds(20,100, 80,30);
8.         value.setBounds(100,100,100,30);
9.         f.add(value); f.add(l1);
10.        f.setSize(300,300);
11.        f.setLayout(null);
12.        f.setVisible(true);
13.    }
14. }
```



Figure 9: PasswordField Example

Java JTextArea

The object of a **JTextArea** class is a multi line region that displays text. It allows the editing of multiple line text. It inherits **JTextComponent** class

public class *JTextArea* **extends** *JTextComponent*

Commonly used Constructors:

Constructor	Description
<code>JTextArea()</code>	Creates a text area that displays no text initially.
<code>JTextArea(String s)</code>	Creates a text area that displays specified text initially.
<code>JTextArea(int row, int column)</code>	Creates a text area with the specified number of rows and columns that displays no text initially.
<code>JTextArea(String s, int row, int column)</code>	Creates a text area with the specified number of rows and columns that displays specified text.

Commonly used Methods:

Methods	Description
void setRows(int rows)	It is used to set specified number of rows.
void setColumns(int cols)	It is used to set specified number of columns.
void setFont(Font f)	It is used to set the specified font.
void insert(String s, int position)	It is used to insert the specified text on the specified position.
void append(String s)	It is used to append the given text to the end of the document.

Example

```
package javaapplication6;
```

```
/**
```

```
*
```

```
* @author BIPIN
```

```
*/
```

```
import javax.swing.*;
```

```
public class TextAreaExample
```

```
{
```

```
    TextAreaExample(){
```

```
        JFrame f= new JFrame();
```

```
        JTextArea area=new JTextArea("Nepal is a Beautiful Country.");
```

```
        area.setBounds(10,30, 200,200);
```

```
        f.add(area);
```

```
        f.setSize(300,300);
```

```
        f.setLayout(null);
```

```
f.setVisible(true);  
}  
public static void main(String args[ ])  
{  
    new TextAreaExample();  
}  
}
```

Output:

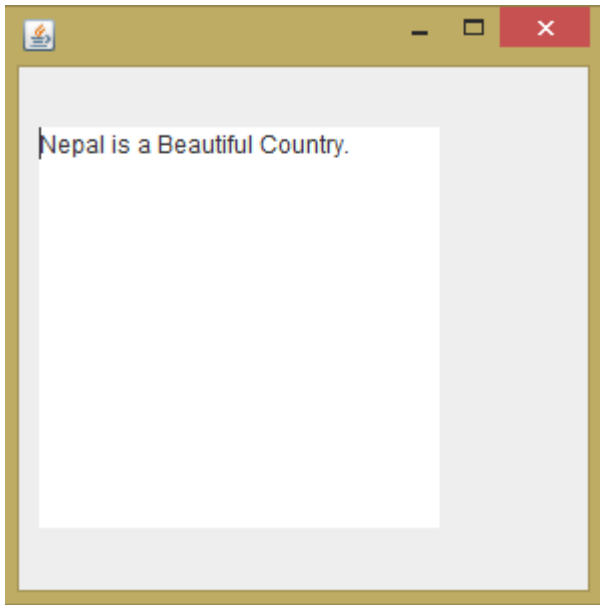


Figure 10:Text Area Example

Java JScrollPane

A **JScrollPane** is used to make scrollable view of a component. When screen size is limited, we use a scroll pane to display a large component or a component whose size can change dynamically.

Constructors

Constructor	Purpose
JScrollPane()	It creates a scroll pane. The Component parameter, when present, sets the scroll pane's client. The two int parameters, when present, set the vertical and horizontal scroll bar policies (respectively).
JScrollPane(Component)	
JScrollPane(int, int)	
JScrollPane(Component, int, int)	

Useful Methods

Modifier	Method	Description
void	setColumnHeaderView(Component)	It sets the column header for the scroll pane.
void	setRowHeaderView(Component)	It sets the row header for the scroll pane.
void	setCorner(String, Component)	It sets or gets the specified corner. The int parameter specifies which corner and must be one of the following constants defined in JScrollPaneConstants: UPPER_LEFT_CORNER, UPPER_RIGHT_CORNER, LOWER_LEFT_CORNER, LOWER_RIGHT_CORNER, LOWER_LEADING_CORNER, LOWER_TRAILING_CORNER, UPPER_LEADING_CORNER, UPPER_TRAILING_CORNER.
Component	getCorner(String)	
void	setViewportView(Component)	Set the scroll pane's client.

JScrollPane Example

```

1. import java.awt.FlowLayout;
2. import javax.swing.JFrame;
3. import javax.swing.JScrollPane;
4. import javax.swing.JTextArea;
5.
6. public class JScrollPaneExample {
7.     private static final long serialVersionUID = 1L;
8.
9.     private static void createAndShowGUI() {
10.
11.         // Create and set up the window.
12.         final JFrame frame = new JFrame("Scroll Pane Example");
13.

```

```

14.    // Display the window.
15.    frame.setSize(500, 500);
16.    frame.setVisible(true);
17.    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18.
19.    // set flow layout for the frame
20.    frame.getContentPane().setLayout(new FlowLayout());
21.
22.    JTextArea textArea = new JTextArea(20, 20);
23.    JScrollPane scrollableTextArea = new JScrollPane(textArea);
24.
25.    scrollableTextArea.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLB
AR_ALWAYS);
26.    scrollableTextArea.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_AL
WAYS);
27.
28.    frame.getContentPane().add(scrollableTextArea);
29. }
30. public static void main(String[] args) {
31.
32.
33.    javax.swing.SwingUtilities.invokeLater(new Runnable() {
34.
35.        public void run() {
36.            createAndShowGUI();
37.        }
38.    });
39. }
40. }

```

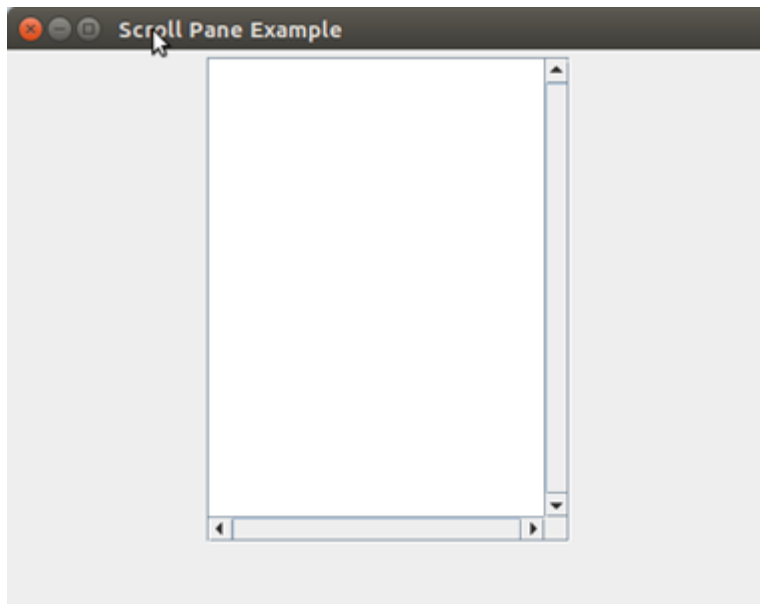


Figure 11: JScrollPane Example

Java JLabel

The object of **JLabel** class is a component for placing text in a container. It is used to display a single line of read only text. The text can be changed by an application but a user cannot edit it directly. It inherits **JComponent** class.

public class JLabel **extends** JComponent **implements** SwingConstants, Accessible

Commonly used Constructors:

Constructor	Description
JLabel()	Creates a JLabel instance with no image and with an empty string for the title.
JLabel(String s)	Creates a JLabel instance with the specified text.
JLabel(Icon i)	Creates a JLabel instance with the specified image.
JLabel(String s, Icon i, int horizontalAlignment)	Creates a JLabel instance with the specified text, image, and horizontal alignment.

Commonly used Methods:

Methods	Description
String getText()	It returns the text string that a label displays.
void setText(String text)	It defines the single line of text this component will display.
void setHorizontalAlignment(int alignment)	It sets the alignment of the label's contents along the X axis.
Icon getIcon()	It returns the graphic image that the label displays.
int getHorizontalAlignment()	It returns the alignment of the label's contents along the X axis.

Example:

```

import javax.swing.*;
class LabelExample
{
    public static void main(String args[])
    {
        JFrame f= new JFrame("Label Example");
        JLabel l1,l2;
        l1=new JLabel("First Label.");
        l1.setBounds(50,50, 100,30);
        l2=new JLabel("Second Label.");
        l2.setBounds(50,100, 100,30);
        f.add(l1); f.add(l2);
        f.setSize(300,300);
        f.setLayout(null);
        f.setVisible(true);
    }
}

```

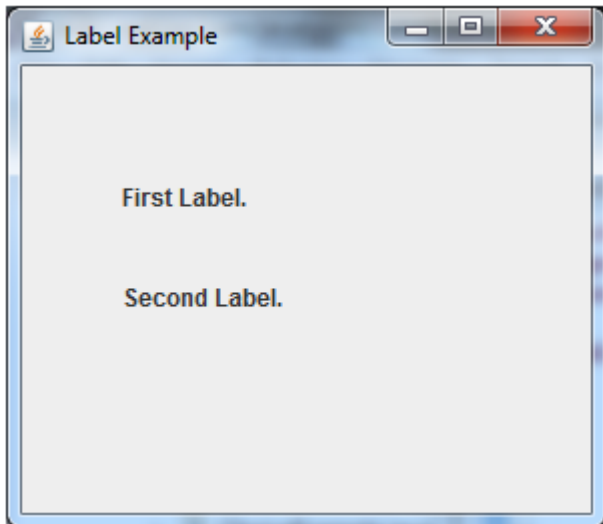


Figure 12: Label Example

Choice Components

- ❖ Check Boxes
- ❖ Radio Buttons
- ❖ Borders
- ❖ Combo boxes
- ❖ Sliders

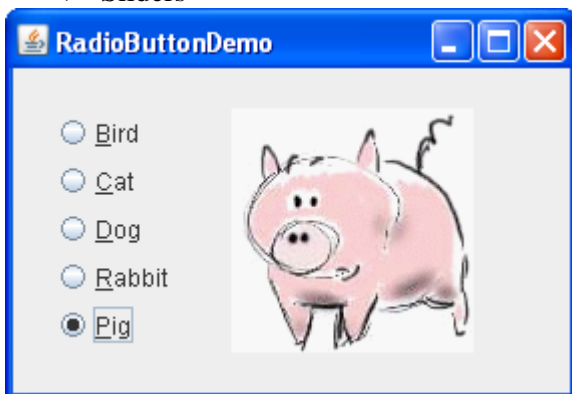


Figure 15: Radio Button

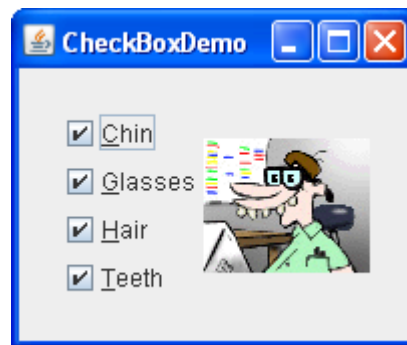


Figure 13: Check Box

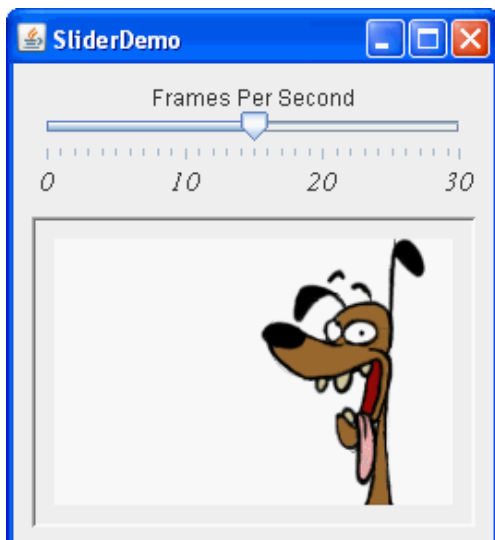


Figure 16: Slider

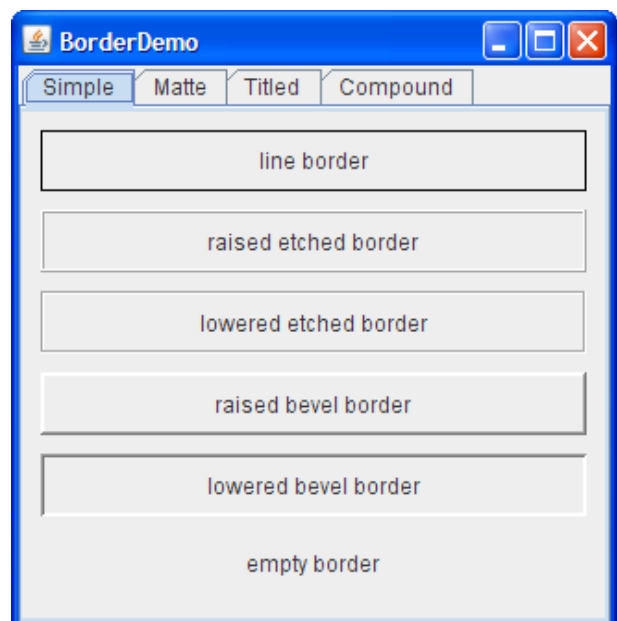


Figure 14: Border

Java JCheckBox

The **JCheckBox** class is used to create a checkbox. It is used to turn an option on (true) or off (false). Clicking on a CheckBox changes its state from "on" to "off" or from "off" to "on ".It inherits JToggleButton class.

public class JCheckBox **extends** JToggleButton **implements** Accessible

Commonly used Constructors:

Constructor	Description
JCheckBox()	Creates an initially unselected check box button with no text, no icon.
JCheckBox(String s)	Creates an initially unselected check box with text.
JCheckBox(String text, boolean selected)	Creates a check box with text and specifies whether or not it is initially selected.
JCheckBox(Action a)	Creates a check box where properties are taken from the Action supplied.

Commonly used Methods:

Methods	Description
AccessibleContext getAccessibleContext()	It is used to get the AccessibleContext associated with this JCheckBox.
protected String paramString()	It returns a string representation of this JCheckBox.

Example

```
import javax.swing.*;
public class CheckBoxExample
{
    CheckBoxExample(){
        JFrame f= new JFrame("CheckBox Example");
        JCheckBox checkBox1 = new JCheckBox("C++");
        checkBox1.setBounds(100,100, 50,50);
        JCheckBox checkBox2 = new JCheckBox("Java", true);
        checkBox2.setBounds(100,150, 50,50);
        f.add(checkBox1);
        f.add(checkBox2);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
    public static void main(String args[])
    {
        new CheckBoxExample();
    }
}
```

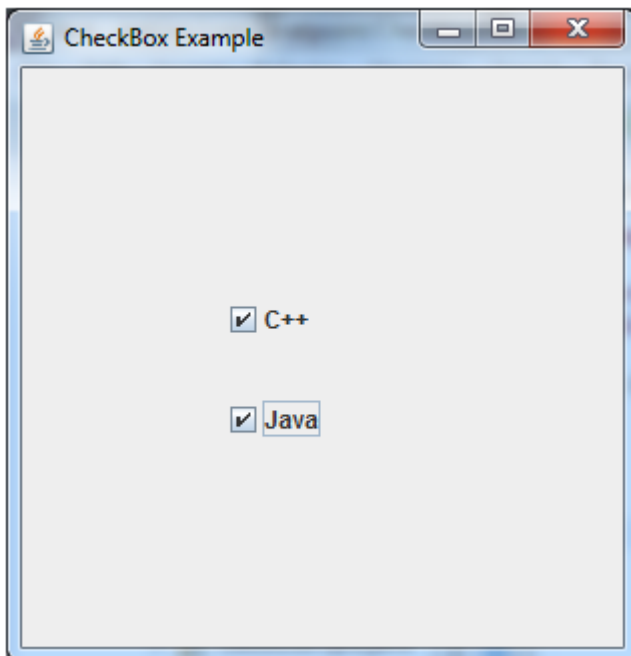


Figure 17: Check Box Example

Java JRadioButton

The **JRadioButton** class is used to create a radio button. It is used to choose one option from multiple options. It is widely used in exam systems or quiz.

It should be added in ButtonGroup to select one radio button only.

public class JRadioButton **extends** JToggleButton **implements** Accessible

Commonly used Constructors:

Constructor	Description
JRadioButton()	Creates an unselected radio button with no text.
JRadioButton(String s)	Creates an unselected radio button with specified text.
JRadioButton(String s, boolean selected)	Creates a radio button with the specified text and selected status.

Commonly used Methods:

Methods	Description
void setText(String s)	It is used to set specified text on button.
String getText()	It is used to return the text of the button.
void setEnabled(boolean b)	It is used to enable or disable the button.
void setIcon(Icon b)	It is used to set the specified Icon on the button.
Icon getIcon()	It is used to get the Icon of the button.
void setMnemonic(int a)	It is used to set the mnemonic on the button.
void addActionListener(ActionListener a)	It is used to add the action listener to this object.

Example


```
import javax.swing.*;

public class RadioButtonExample {
    JFrame f;

    RadioButtonExample(){
        f=new JFrame();
        JRadioButton r1=new JRadioButton("A) Male");
        JRadioButton r2=new JRadioButton("B) Female");
        r1.setBounds(75,50,100,30);
        r2.setBounds(75,100,100,30);
        ButtonGroup bg=new ButtonGroup();
        bg.add(r1);bg.add(r2);
        f.add(r1);f.add(r2);
        f.setSize(300,300);
        f.setLayout(null);
        f.setVisible(true);
    }

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

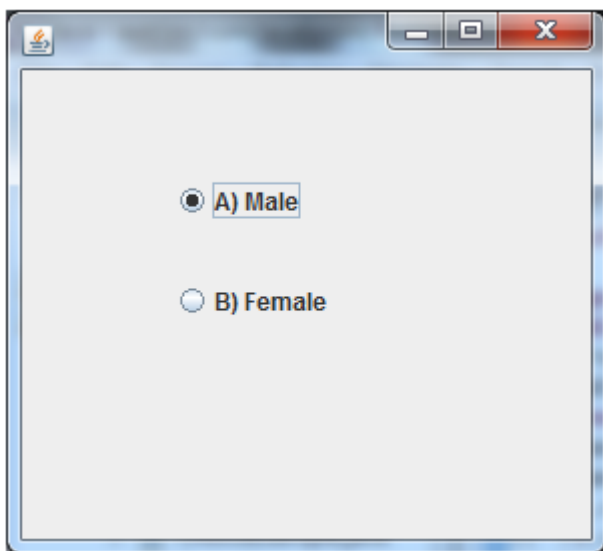


Figure 18: Radio Button Example

Java JComboBox

The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a menu. It inherits JComponent class.

public class JComboBox **extends** JComponent **implements** ItemSelectable, ListDataListener, ActionListener, Accessible

Commonly used Constructors:

Constructor	Description
JComboBox()	Creates a JComboBox with a default data model.
JComboBox(Object[] items)	Creates a JComboBox that contains the elements in the specified array.
JComboBox(Vector<?> items)	Creates a JComboBox that contains the elements in the specified Vector.

Commonly used Methods:

Methods	Description
void addItem(Object anObject)	It is used to add an item to the item list.
void removeItem(Object anObject)	It is used to delete an item to the item list.
void removeAllItems()	It is used to remove all the items from the list.
void setEditable(boolean b)	It is used to determine whether the JComboBox is editable.
void addActionListener(ActionListener a)	It is used to add the ActionListener.
void addItemListener(ItemListener i)	It is used to add the ItemListener.

```

package javaapplication6;

/**
 *
 * @author BIPIN
 */
import javax.swing.*;

public class ComboBoxExample {
    JFrame f;

    ComboBoxExample(){
        f=new JFrame("ComboBox Example");
        String
subjects[]={ "ADBMS", "IT", "NSA", "AdvancedJava", "DBA"};
        JComboBox cb=new JComboBox(subjects);
        cb.setBounds(50, 50,90,20);
        f.add(cb);
        f.setLayout(null);
        f.setSize(400,500);
        f.setVisible(true);
    }

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

```

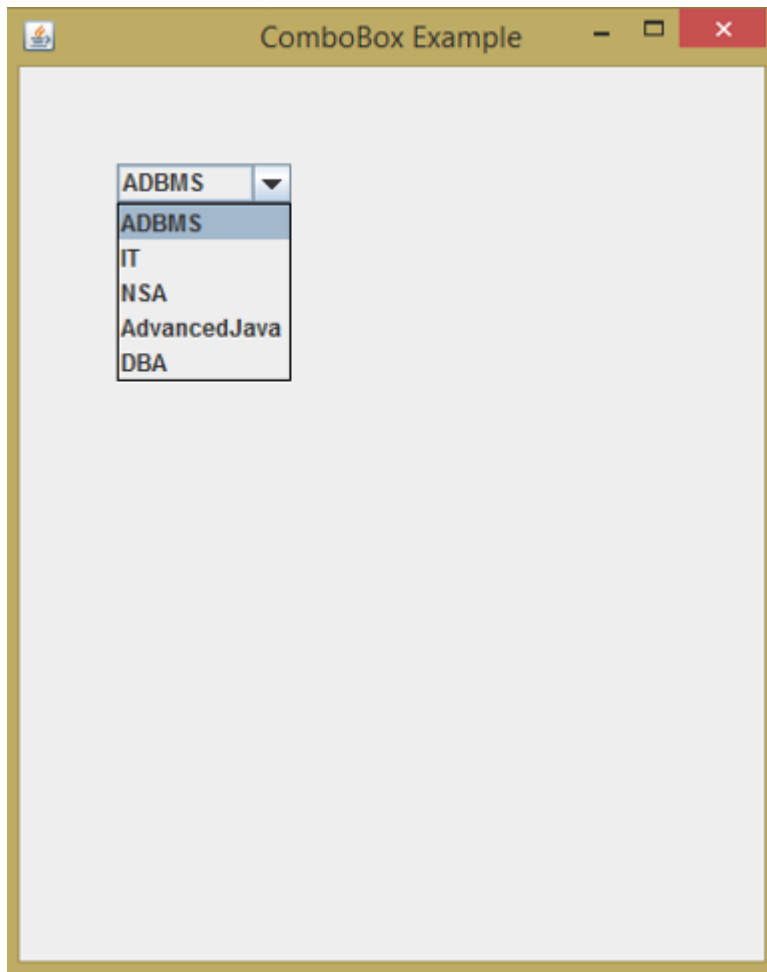


Figure 19: Combo Box Example

Java JSlider

The Java **JSlider** class is used to create the slider. By using JSlider, a user can select a value from a specific range.

Commonly used Constructors of JSlider class

Constructor	Description
JSlider()	creates a slider with the initial value of 50 and range of 0 to 100.
JSlider(int orientation)	creates a slider with the specified orientation set by either JSlider.HORIZONTAL or JSlider.VERTICAL with the range 0 to 100 and initial value 50.
JSlider(int min, int max)	creates a horizontal slider using the given min and max.

JSlider(int min, int max, int value)	creates a horizontal slider using the given min, max and value.
JSlider(int orientation, int min, int max, int value)	creates a slider using the given orientation, min, max and value.

Commonly used Methods of JSlider class

Method	Description
public void setMinorTickSpacing(int n)	is used to set the minor tick spacing to the slider.
public void setMajorTickSpacing(int n)	is used to set the major tick spacing to the slider.
public void setPaintTicks(boolean b)	is used to determine whether tick marks are painted.
public void setPaintLabels(boolean b)	is used to determine whether labels are painted.
public void setPaintTracks(boolean b)	is used to determine whether track is painted.

Example

```
import javax.swing.*;
public class SliderExample1 extends JFrame{
    public SliderExample1() {
        JSlider slider = new JSlider(JSlider.HORIZONTAL, 0, 50, 25);
        JPanel panel=new JPanel();
        panel.add(slider);
        add(panel);
    }

    public static void main(String s[]) {
        SliderExample1 frame=new SliderExample1();
        frame.pack();
    }
}
```

```

frame.setVisible(true);
}
}

```

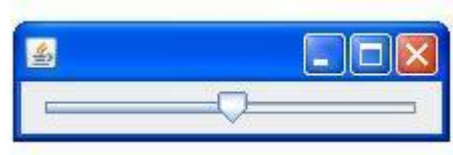


Figure 20: Slider Example

Borders

Every **JComponent** can have one or more borders. Borders are incredibly useful objects that, while not themselves components, know how to draw the edges of Swing components. Borders are useful not only for drawing lines and fancy edges, but also for providing titles and empty space around components.

To put a border around a **JComponent**, you use its **setBorder** method. You can use the **BorderFactory** class to create most of the borders that Swing provides.

Example of different types of borders

```

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Container;
import java.awt.GridLayout;

import javax.swing.JApplet;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.border.BevelBorder;
import javax.swing.border.Border;
import javax.swing.border.CompoundBorder;
import javax.swing.border.EtchedBorder;
import javax.swing.border.LineBorder;
import javax.swing.border.MatteBorder;
import javax.swing.border.SoftBevelBorder;
import javax.swing.border.TitledBorder;

public class Borders extends JApplet {
    static JPanel showBorder(Border b) {
        JPanel jp = new JPanel();
        jp.setLayout(new BorderLayout());
        String nm = b.getClass().toString();
        nm = nm.substring(nm.lastIndexOf('.') + 1);
    }
}

```

```

        jp.add(new JLabel(nm, JLabel.CENTER), BorderLayout.CENTER);
        jp.setBorder(b);
        return jp;
    }

    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new GridLayout(2, 4));
        cp.add(showBorder(new TitledBorder("Title")));
        cp.add(showBorder(new EtchedBorder()));
        cp.add(showBorder(new LineBorder(Color.BLUE)));
        cp.add(showBorder(new MatteBorder(5, 5, 30, 30, Color.GREEN)));
        cp.add(showBorder(new BevelBorder(BevelBorder.RAISED)));
        cp.add(showBorder(new SoftBevelBorder(BevelBorder.LOWERED)));
        cp.add(showBorder(new CompoundBorder(new EtchedBorder(),
            new LineBorder(Color.RED))));
    }

    public static void main(String[] args) {
        run(new Borders(), 500, 300);
    }

    public static void run(JApplet applet, int width, int height) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(applet);
        frame.setSize(width, height);
        applet.init();
        applet.start();
        frame.setVisible(true);
    }
}

```

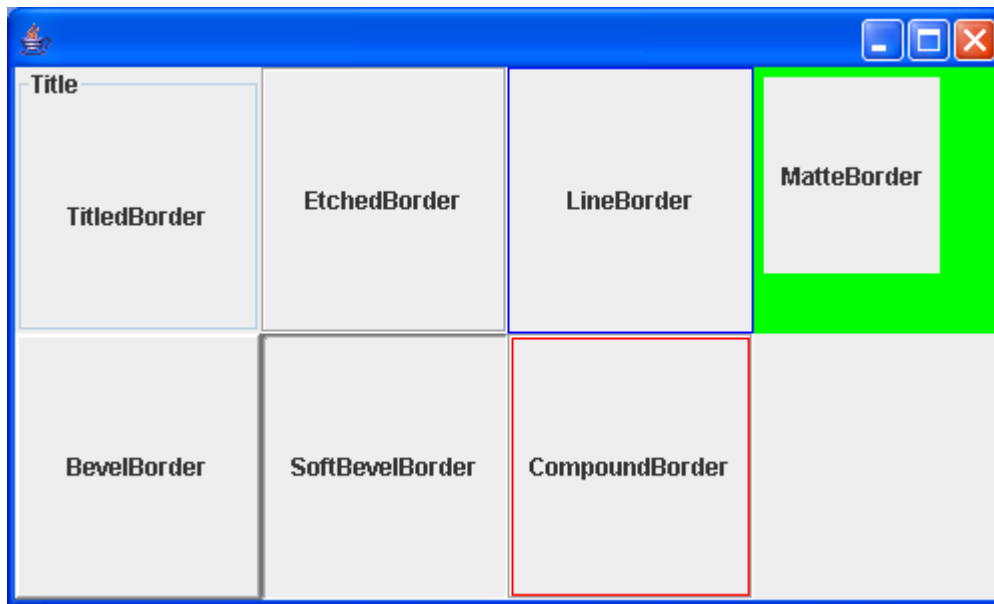


Figure 21: Example of different types of border

Menus

A menu provides a space-saving way to let the user choose one of several options. Other components with which the user can make a one-of-many choice include combo boxes, lists, radio buttons, spinners, and tool bars.

Menus are unique in that, by convention, they aren't placed with the other components in the UI. Instead, a menu usually appears either in a menu bar or as a popup menu. A menu bar contains one or more menus and has a customary, platform-dependent location — usually along the top of a window. A popup menu is a menu that is invisible until the user makes a platform-specific mouse action, such as pressing the right mouse button, over a popup-enabled component. The popup menu then appears under the cursor.

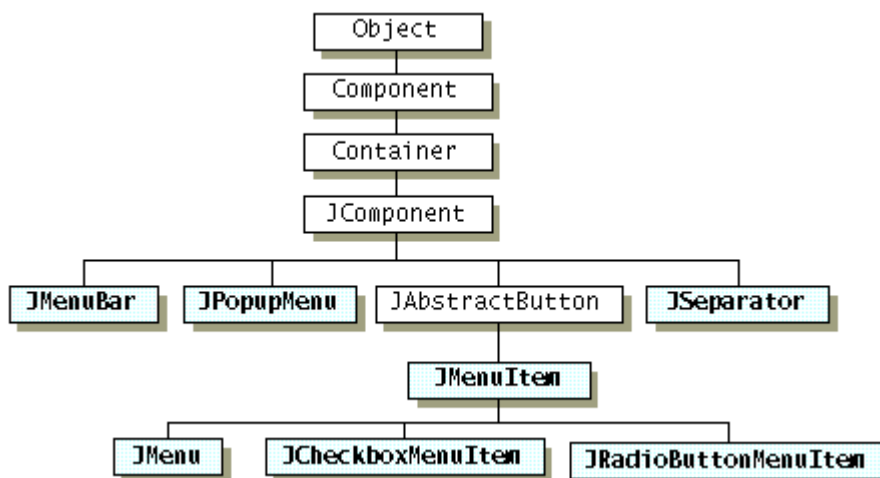


Figure 22: The Menu Component Hierarchy

The following figure shows many menu-related components: a menu bar, menus, menu items, radio button menu items, check box menu items, and separators. As you can see, a menu item can have either an image or text, or both. You can also specify other properties, such as font and color.

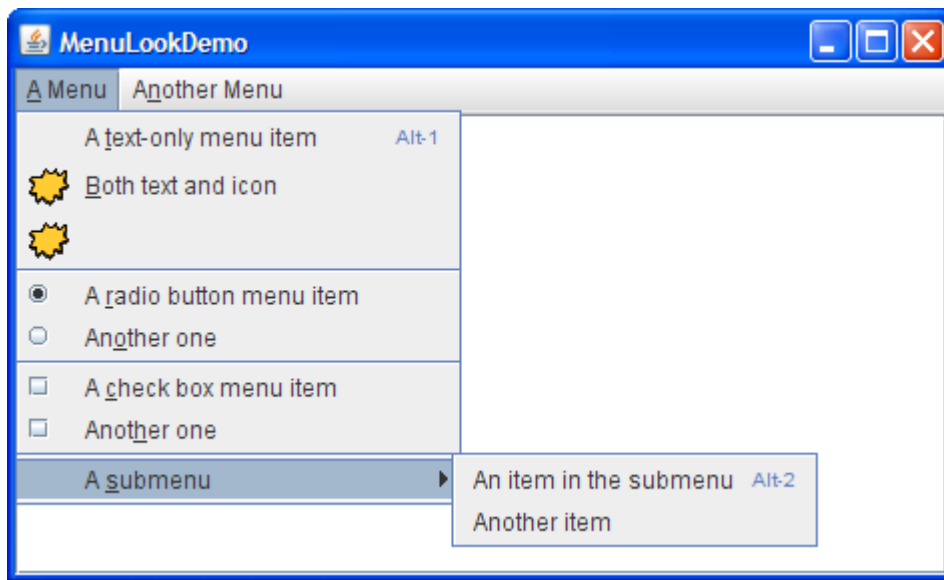


Figure 23: Menus

Java JMenuBar, JMenu and JMenuItem

The **JMenuBar** class is used to display menu bar on the window or frame. It may have several menus.

The object of **JMenu** class is a pull down menu component which is displayed from the menu bar. It inherits the **JMenuItem** class.

The object of **JMenuItem** class adds a simple labeled menu item. The items used in a menu must belong to the **JMenuItem** or any of its subclass.

```
public class JMenuBar extends JComponent implements MenuElement, Accessible
```

```
public class JMenu extends JMenuItem implements MenuElement, Accessible
```

```
public class JMenuItem extends AbstractButton implements Accessible, MenuElement
```

Example

```

import javax.swing.*;
class MenuExample
{
    JMenu menu, submenu;
    JMenuItem i1, i2, i3, i4, i5;
    MenuExample(){
        JFrame f= new JFrame("Menu and MenuItem Example");
        JMenuBar mb=new JMenuBar();
        menu=new JMenu("Menu");
        submenu=new JMenu("Sub Menu");
        i1=new JMenuItem("Item 1");
        i2=new JMenuItem("Item 2");
        i3=new JMenuItem("Item 3");
        i4=new JMenuItem("Item 4");
        i5=new JMenuItem("Item 5");
        menu.add(i1); menu.add(i2); menu.add(i3);
        submenu.add(i4); submenu.add(i5);
        menu.add(submenu);
        mb.add(menu);
        f.setJMenuBar(mb);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
    public static void main(String args[])
    {
        new MenuExample();
    }
}

```

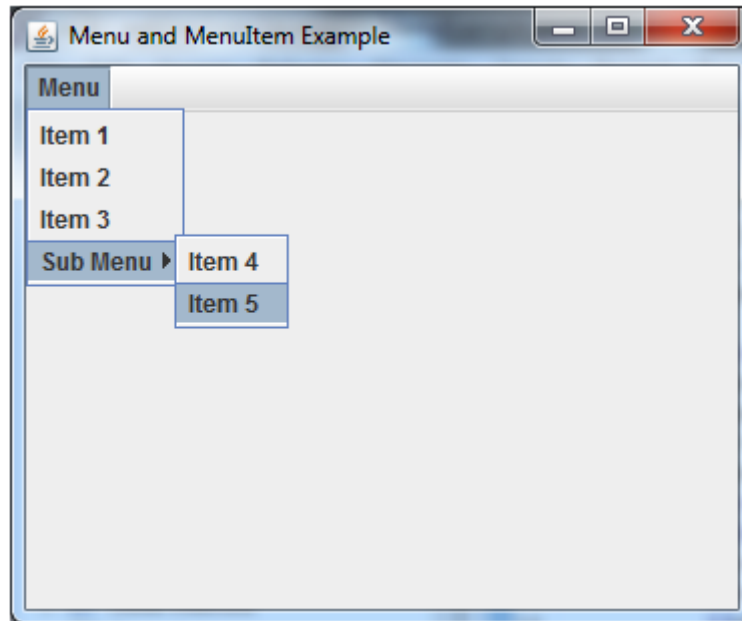


Figure 24: Menu Example

- Swing provides support for pull down and popup menus.
- A **JMenuBar** can contain several **JMenu**s.
- Each of the **JMenu**s can contain a series of **MenuItem**s that we can select.

Creating Menu

- First, A **JMenuBar** is created
- Then, we attach all of the menus to this **JMenuBar**
- Then we add **MenuItem** to the **JMenu**
- The **JMenuBar** is then added to the frame by using **setJMenuBar()** method

By default , each **MenuItem** added to a **JMenu** is enabled – that is, it can be selected. In certain situations, we may need to disable a **MenuItem**. This is done by calling *setEnabled(false)*.

Icons in Menu

Many Swing components, such as labels, buttons, and tabbed panes, can be decorated with an *icon* — a *fixed-sized picture*. An icon is an object that adheres to the **Icon** interface. Swing provides a particularly useful implementation of the **Icon** interface: **ImageIcon**, which paints an icon from a GIF, JPEG, or PNG image.

Icon can be added in menu by following constructors :

MenuItem(Icon icon)

Creates a **MenuItem** with the specified icon.

MenuItem(String text, Icon icon)

Creates a **MenuItem** with the specified text and icon.

Example (not runnable full code just important lines!!)

```
ImageIcon sampleIcon = new ImageIcon("src/main/resources/sample.png");
JMenu sampleMenu = new JMenu("Sample");
JMenuItem eMenuItem = new JMenuItem("Sample", sampleIcon);
sampleMenu.add(eMenuItem);
```

Java JPopupMenu

PopupMenu can be dynamically popped up at specific position within a component. It inherits the JComponent class.

public class JPopupMenu **extends** JComponent **implements** Accessible, MenuElement

Commonly used Constructors:

Constructor	Description
JPopupMenu()	Constructs a JPopupMenu without an "invoker".
JPopupMenu(String label)	Constructs a JPopupMenu with the specified title.

Example:

```
import javax.swing.*;
import java.awt.event.*;
class PopupMenuExample
{
    PopupMenuExample(){
        final JFrame f= new JFrame("PopupMenu Example");
        final JPopupMenu popupmenu = new JPopupMenu("Edit");
        JMenuItem cut = new JMenuItem("Cut");
        JMenuItem copy = new JMenuItem("Copy");
        JMenuItem paste = new JMenuItem("Paste");
        popupmenu.add(cut); popupmenu.add(copy); popupmenu.add(paste);

        f.addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                popupmenu.show(f , e.getX(), e.getY());
            }
        });
    }
}
```

```
f.add(popupmenu);  
f.setSize(300,300);  
f.setLayout(null);  
f.setVisible(true);  
}  
public static void main(String args[])  
{  
    new PopupMenuExample();  
}}
```

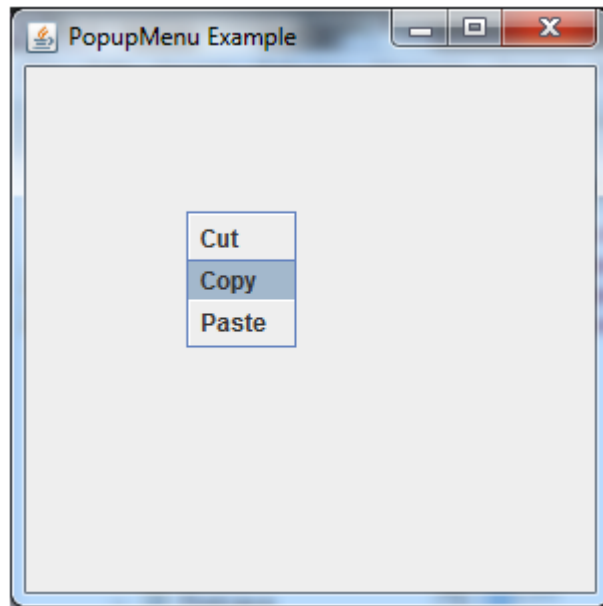


Figure 25: Popup Menu Example

Checkbox in menu item

The **CheckboxMenuItem** class represents a check box which can be included in a menu. Selecting the check box in the menu changes control's state from on to off or from off to on.

JCheckBoxMenuItem class represents checkbox which can be included on a menu . A CheckBoxMenuItem can have text or a graphic icon or both, associated with it. MenuItem can be selected or deselected. MenuItems can be configured and controlled by actions.

Nested class

Modifier and Type	Class	Description
protected class	JCheckBoxMenuItem.AccessibleJCheckBoxMenuItem	This class implements accessibility support for the JCheckBoxMenuItem class.

Constructor

Constructor	Description
JCheckBoxMenuItem()	It creates an initially unselected check box menu item with no set text or icon.
JCheckBoxMenuItem(Action a)	It creates a menu item whose properties are taken from the Action supplied.
JCheckBoxMenuItem(Icon icon)	It creates an initially unselected check box menu item with an icon.
JCheckBoxMenuItem(String text)	It creates an initially unselected check box menu item with text.
JCheckBoxMenuItem(String text, boolean b)	It creates a check box menu item with the specified text and selection state.

JCheckBoxMenuItem(String text, Icon icon)	It creates an initially unselected check box menu item with the specified text and icon.
JCheckBoxMenuItem(String text, Icon icon, boolean b)	It creates a check box menu item with the specified text, icon, and selection state.

Methods

Modifier	Method	Description
AccessibleContext	getAccessibleContext()	It gets the AccessibleContext associated with this JCheckBoxMenuItem.
Object[]	getSelectedObjects()	It returns an array (length 1) containing the check box menu item label or null if the check box is not selected.
boolean	getState()	It returns the selected-state of the item.
String	getUIClassID()	It returns the name of the L&F class that renders this component.
protected String	paramString()	It returns a string representation of this JCheckBoxMenuItem.
void	setState(boolean b)	It sets the selected-state of the item.

Example:

(Event handling part will be discussed in next unit!)

1. **import** java.awt.event.ActionEvent;
2. **import** java.awt.event.ActionListener;
3. **import** java.awt.event.KeyEvent;
4. **import** javax.swing.AbstractButton;
5. **import** javax.swing.Icon;
6. **import** javax.swing.JCheckBoxMenuItem;
7. **import** javax.swing.JFrame;
8. **import** javax.swing.JMenu;

```

9. import javax.swing.JMenuBar;
10. import javax.swing.JMenuItem;
11.
12. public class JavaCheckBoxMenuItem {
13.     public static void main(final String args[]) {
14.         JFrame frame = new JFrame("Jmenu Example");
15.         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16.         JMenuBar menuBar = new JMenuBar();
17.         // File Menu, F - Mnemonic
18.         JMenu fileMenu = new JMenu("File");
19.         fileMenu.setMnemonic(KeyEvent.VK_F);
20.         menuBar.add(fileMenu);
21.         // File->New, N - Mnemonic
22.         JMenuItem menuItem1 = new JMenuItem("Open", KeyEvent.VK_N);
23.         fileMenu.add(menuItem1);
24.
25.         JCheckBoxMenuItem caseMenuItem = new JCheckBoxMenuItem("Option_1");
26.         caseMenuItem.setMnemonic(KeyEvent.VK_C);
27.         fileMenu.add(caseMenuItem);
28.
29.         ActionListener aListener = new ActionListener() {
30.             public void actionPerformed(ActionEvent event) {
31.                 AbstractButton aButton = (AbstractButton) event.getSource();
32.                 boolean selected = aButton.getModel().isSelected();
33.                 String newLabel;
34.                 Icon newIcon;
35.                 if (selected) {
36.                     newLabel = "Value-1";
37.                 } else {
38.                     newLabel = "Value-2";
39.                 }
40.                 aButton.setText(newLabel);
41.             }
42.         };
43.
44.         caseMenuItem.addActionListener(aListener);
45.         frame.setJMenuBar(menuBar);
46.         frame.setSize(350, 250);
47.         frame.setVisible(true);
48.     }

```


49. }

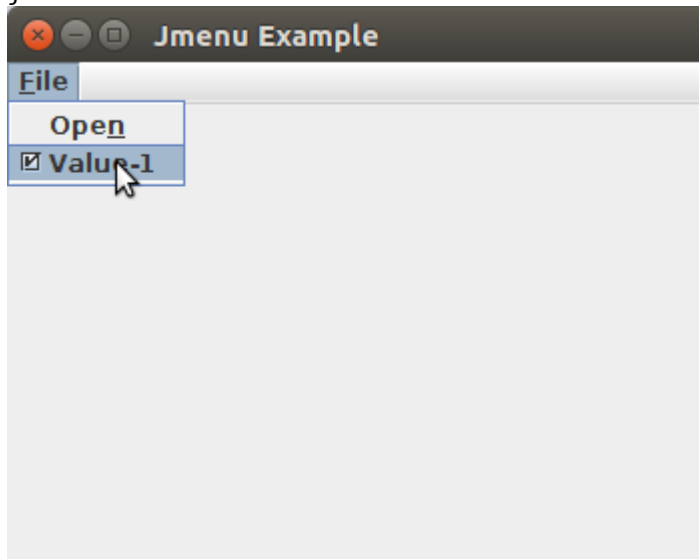


Figure 26: Jmenu Example

Radio Button in Menu

`javax.swing.JRadioButtonMenuItem` class can be used.

```
public class JRadioButtonMenuItem
    extends JMenuItem
        implements Accessible
```

Class Constructors

Sr.No.	Constructor & Description
1	JRadioButtonMenuItem() Creates a JRadioButtonMenuItem with no set text or icon.
2	JRadioButtonMenuItem(Action a) Creates a radio button menu item whose properties are taken from the Action supplied.
3	JRadioButtonMenuItem(Icon icon) Creates a JRadioButtonMenuItem with an icon.
4	JRadioButtonMenuItem(Icon icon, boolean selected) Creates a radio button menu item with the specified image and selection state, but no text.

5	JRadioButtonMenuItem(String text) Creates a JRadioButtonMenuItem with text.
6	JRadioButtonMenuItem(String text, boolean selected) Creates a radio button menu item with the specified text and selection state.
7	JRadioButtonMenuItem(String text, Icon icon) Creates a radio button menu item with the specified text and Icon.
8	JRadioButtonMenuItem(String text, Icon icon, boolean selected) Creates a radio button menu item with the specified text, image, and selection state.

Class Methods

Sr.No.	Method & Description
1	AccessibleContext getAccessibleContext() Gets the AccessibleContext associated with this JRadioButtonMenuItem.
2	String getUIClassID() Returns the name of the L&F class that renders this component.
3	protected String paramString() Returns a string representation of this JRadioButtonMenuItem.

Methods Inherited

This class inherits methods from the following classes –

javax.swing.JMenuItem
javax.swing.JAbstractButton
javax.swing.JComponent
java.awt.Container
java.awt.Component
java.lang.Object

Setting Mnemonics, Accelerators and Tooltip Text

- Menu support two kinds of keyboard alternatives: *mnemonics* and *accelerators*.
 - Mnemonics offer a way to use the keyboard to navigate the menu hierarchy and thus increases accessibility to programs.
 - A mnemonic is a key that makes an already visible menu item be chosen.
 - A menu item generally display its mnemonics by underlining the first occurrence of the mnemonics character in the menu's text.
 - We can specify mnemonics for menu items by using **setMnemonics()** method.
- Accelerators on the other hand, offer keyboard shortcuts to bypass navigating the menu hierarchy.
- An accelerator is a key combination that causes a menu item to be chosen, whether or not it's visible.
- To specify an accelerator can use **setAccelerator()** method.

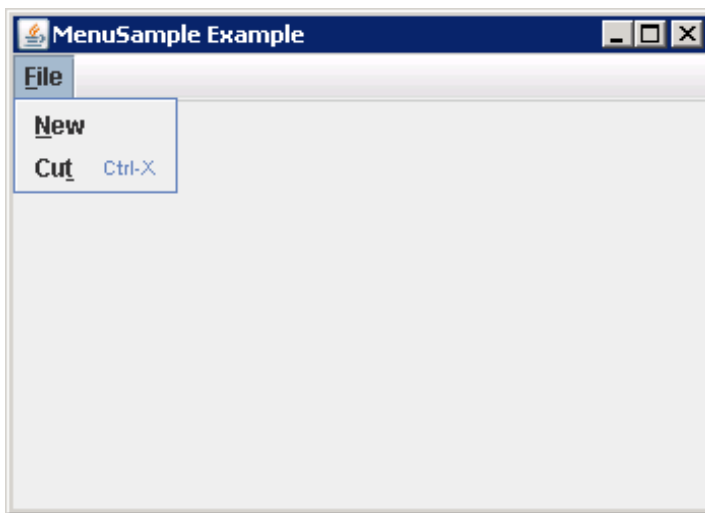


Figure 27: Accelerators & Menu

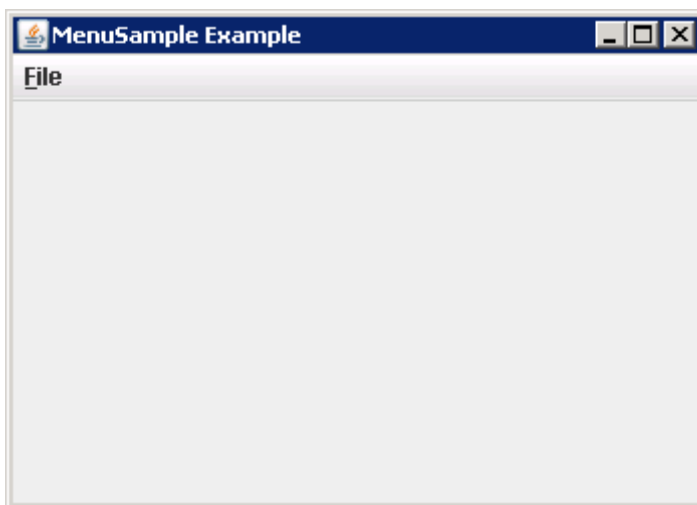


Figure 28: Menu & Mnemonics

- Tooltip texts are very useful because they explain to the users how to use specific component.
- We use **setToolTipText()** method to specify the text to display in the tooltip.
- The text gets displayed when the cursor lingers over the component.
- We can use **getToolTipText()** method to get the tooltip text.

Example : ToolTipText

```
import javax.swing.*;

public class ToolTipExample {
    public static void main(String[] args) {
        JFrame f=new JFrame("Password Field Example");
        //Creating PasswordField and label
        JPasswordField value = new JPasswordField();
        value.setBounds(100,100,100,30);
        value.setToolTipText("Enter your Password");
        JLabel l1=new JLabel("Password:");
        l1.setBounds(20,100, 80,30);
        //Adding components to frame
        f.add(value); f.add(l1);
        f.setSize(300,300);
        f.setLayout(null);
        f.setVisible(true);
    }
}
```



Figure 29: ToolTipText Example

Example : Mnemonics and Accelerator

```

import java.awt.event.KeyEvent;

import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.KeyStroke;

public class JMenuItemKeyStroke {

    public static void main(final String args[]) {
        JFrame frame = new JFrame("MenuSample Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JMenuBar menuBar = new JMenuBar();

        // File Menu, F - Mnemonic
        JMenu fileMenu = new JMenu("File");
        fileMenu.setMnemonic(KeyEvent.VK_F);
        menuBar.add(fileMenu);

        // File->New, N - Mnemonic
        JMenuItem newItem = new JMenuItem("New", KeyEvent.VK_N);
        fileMenu.add(newItem);

        // Edit->Cut, T - Mnemonic, CTRL-X - Accelerator
        JMenuItem cutMenuItem = new JMenuItem("Cut", KeyEvent.VK_T);
        KeyStroke ctrlXKeyStroke = KeyStroke.getKeyStroke("control X");
        cutMenuItem.setAccelerator(ctrlXKeyStroke);
        fileMenu.add(cutMenuItem);
        frame.setJMenuBar(menuBar);
        frame.setSize(350, 250);
        frame.setVisible(true);
    }
}

```

Enabling and Disabling Menu

All Swing components have a **setEnabled(boolean)** method that can enable and disable the component. Disabled components are usually displayed in light gray and do not react to input..

By default , each **JMenuItem** added to a **JMenu** is enabled – that is, it can be selected. In certain situations, we may need to disable a JMenuItem. This is done by calling *setEnabled(false)*.

Java JToolBar

JToolBar container allows us to group other components, usually buttons with icons in a row or column. **JToolBar** provides a component which is useful for displaying commonly used actions or controls.

Nested Classes

Modifier and Type	Class	Description
protected class	JToolBar.AccessibleJToolBar	This class implements accessibility support for the JToolBar class.
static class	JToolBar.Separator	A toolbar-specific separator.

Constructors

Constructor	Description
JToolBar()	It creates a new tool bar; orientation defaults to HORIZONTAL.
JToolBar(int orientation)	It creates a new tool bar with the specified orientation.
JToolBar(String name)	It creates a new tool bar with the specified name.
JToolBar(String name, int orientation)	It creates a new tool bar with a specified name and orientation.

Useful Methods

Modifier and Type	Method	Description
JButton	add(Action a)	It adds a new JButton which dispatches the action.
protected void	addImpl(Component comp, Object constraints, int index)	If a JButton is being added, it is initially set to be disabled.
Void	addSeparator()	It appends a separator of default size to the end of the tool bar.
protected PropertyChangeListener	createActionChangeListener(JButton b)	It returns a properly configured PropertyChangeListener which updates the control as changes to the Action occur, or null if the default property change listener for the control is desired.
protected JButton	createActionComponent(Action a)	Factory method which creates the JButton for Actions added to the JToolBar.
ToolBarUI	getUI()	It returns the tool bar's current UI.

Void	setUI(ToolBarUI ui)	It sets the L&F object that renders this component.
Void	setOrientation(int o)	It sets the orientation of the tool bar.

Java JToolBar Example

```

1. import java.awt.BorderLayout;
2. import java.awt.Container;
3. import javax.swing.JButton;
4. import javax.swing.JComboBox;
5. import javax.swing.JFrame;
6. import javax.swing.JScrollPane;
7. import javax.swing.JTextArea;
8. import javax.swing.JToolBar;
9.
10. public class JToolBarExample {
11.     public static void main(final String args[]) {
12.         JFrame myframe = new JFrame("JToolBar Example");
13.         myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14.         JToolBar toolbar = new JToolBar();
15.         toolbar.setRollover(true);
16.         JButton button = new JButton("File");
17.         toolbar.add(button);
18.         toolbar.addSeparator();
19.         toolbar.add(new JButton("Edit"));
20.         toolbar.add(new JComboBox(new String[] { "Opt-1", "Opt-2", "Opt-3", "Opt-4" }));
21.         Container contentPane = myframe.getContentPane();
22.         contentPane.add(toolbar, BorderLayout.NORTH);
23.         JTextArea textArea = new JTextArea();
24.         JScrollPane mypane = new JScrollPane(textArea);
25.         contentPane.add(mypane, BorderLayout.EAST);
26.         myframe.setSize(450, 250);
27.         myframe.setVisible(true);
28.     }
29. }

```

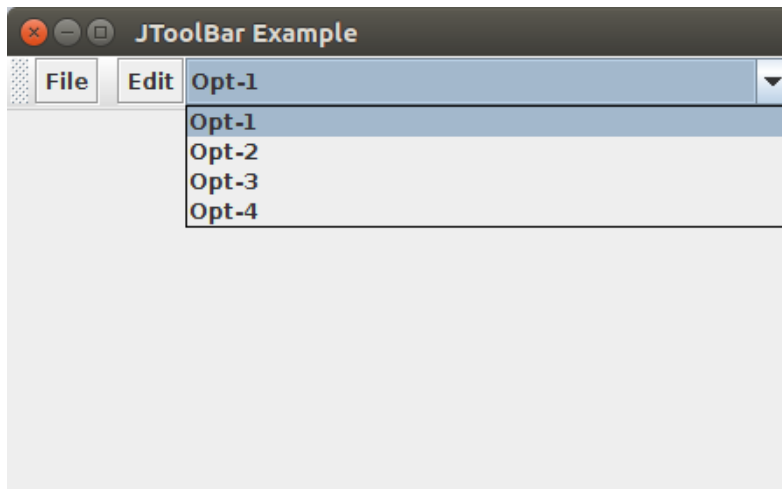



Figure 30: Toolbar Example

Dialog Boxes

A Dialog window is an independent sub window meant to carry temporary notice apart from the main Swing Application Window. Most Dialogs present an error message or warning to a user, but Dialogs can present images, directory trees, or just about anything compatible with the main Swing Application that manages them.

For convenience, several Swing component classes can directly instantiate and display dialogs. To create simple, standard dialogs, you use the **JOptionPane** class. The **ProgressMonitor** class can put up a dialog that shows the progress of an operation. Two other classes, **JColorChooser** and **JFileChooser**, also supply standard dialogs. To bring up a print dialog, you can use the **Printing** API. To create a custom dialog, use the **JDialog** class directly.

Java JDialog

The **JDialog** control represents a top level window with a border and a title used to take some form of input from the user. It inherits the **Dialog** class.

- Unlike JFrame, it doesn't have maximize and minimize buttons.
- declaration for javax.swing.JDialog class:

```
public class JDialog extends Dialog implements WindowConstants, Accessible,
RootPaneContainer
```









Commonly used Constructors:

Constructor	Description
JDialog()	It is used to create a modeless dialog without a title and without a specified Frame owner.
JDialog(Frame owner)	It is used to create a modeless dialog with specified Frame as its owner and an empty title.
JDialog(Frame owner, String title, boolean modal)	It is used to create a dialog with the specified title, owner Frame and modality.

Java JOptionPane

The **JOptionPane** class is used to provide standard dialog boxes such as message dialog box, confirm dialog box and input dialog box. These dialog boxes are used to display information or get input from the user. The JOptionPane class inherits JComponent class.

Using JOptionPane, you can quickly create and customize several different kinds of dialogs. JOptionPane provides support for laying out standard dialogs, providing icons, specifying the dialog title and text, and customizing the button text. Other features allow you to customize the components the dialog displays and specify where the dialog should appear onscreen

icons used by JOptionPane		
Icon description	Java look and feel	Windows look and feel
question		
information		
warning		
error		

Declaration of class:

public class JOptionPane **extends** JComponent **implements** Accessible

Common Constructors of JOptionPane class

Constructor	Description
JOptionPane()	It is used to create a JOptionPane with a test message.
JOptionPane(Object message)	It is used to create an instance of JOptionPane to display a message.
JOptionPane(Object message, int messageType)	It is used to create an instance of JOptionPane to display a message with specified message type and default options.

Common Methods of JOptionPane class

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

Java JOptionPane Example: showMessageDialog()

```
import javax.swing.*;

public class OptionPaneExample {

    JFrame f;

    OptionPaneExample(){

        f=new JFrame();

        JOptionPane.showMessageDialog(f,"Hello, We are BScCSIT 7th sem.");

    }

    public static void main(String[] args) {

        new OptionPaneExample();

    }

}
```

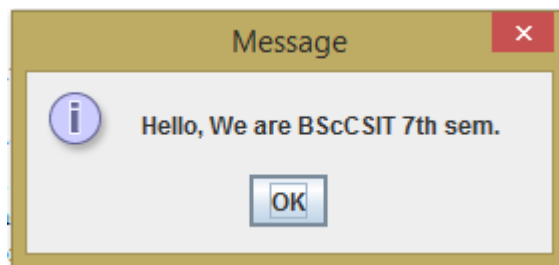
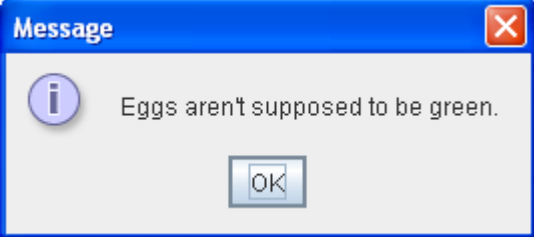
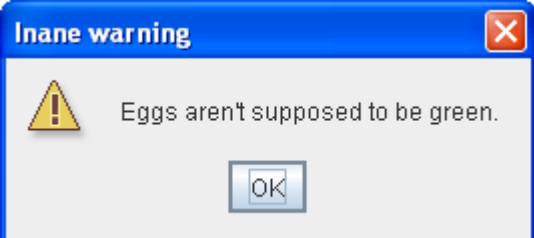
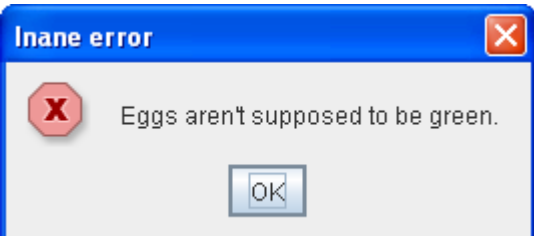
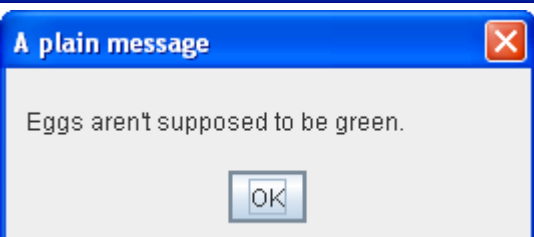
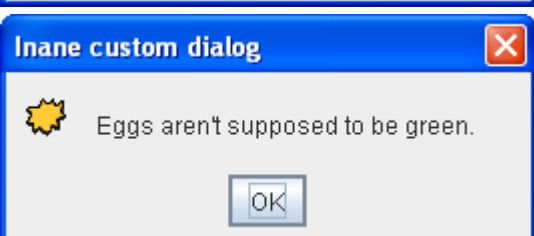


Figure 31: Option Pane Example

showMessageDialog: Displays a modal dialog with one button, which is labeled "OK" (or the localized equivalent). You can easily specify the message, icon, and title that the dialog displays. Here are some examples of using showMessageDialog:

	<pre>//default title and icon JOptionPane.showMessageDialog(frame, "Eggs are not supposed to be green.");</pre>
	<pre>//custom title, warning icon JOptionPane.showMessageDialog(frame, "Eggs are not supposed to be green.", "Inane warning", JOptionPane.WARNING_MESSAGE);</pre>
	<pre>//custom title, error icon JOptionPane.showMessageDialog(frame, "Eggs are not supposed to be green.", "Inane error", JOptionPane.ERROR_MESSAGE);</pre>
	<pre>//custom title, no icon JOptionPane.showMessageDialog(frame, "Eggs are not supposed to be green.", "A plain message", JOptionPane.PLAIN_MESSAGE);</pre>
	<pre>//custom title, custom icon JOptionPane.showMessageDialog(frame, "Eggs are not supposed to be green.", "Inane custom dialog", JOptionPane.INFORMATION_MESSAGE, icon);</pre>

Java JOptionPane Example: showInputDialog()

```

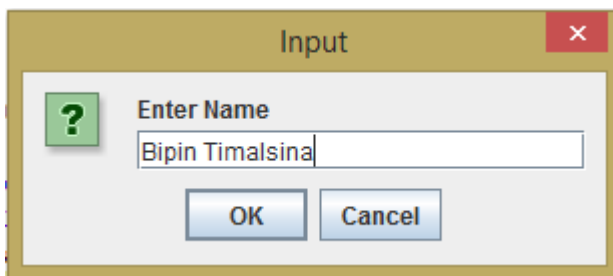
import javax.swing.*;

public class OptionPaneExample {
    JFrame f;

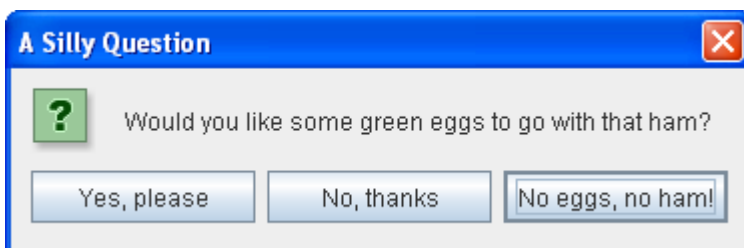
    OptionPaneExample(){
        f=new JFrame();
        String name=JOptionPane.showInputDialog(f,"Enter Name");
    }

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

```

*Figure 32: Input Dialog Example***Java JOptionPane Example: showOptionDialog()**

Displays a modal dialog with the specified buttons, icons, message, title, and so on. With this method, you can change the text that appears on the buttons of standard dialogs. You can also perform many other kinds of customization.

*Figure 33:Option Dialog Example*

```

//Custom button text
Object[] options = {"Yes, please",
                    "No, thanks",
                    "No eggs, no ham!"};

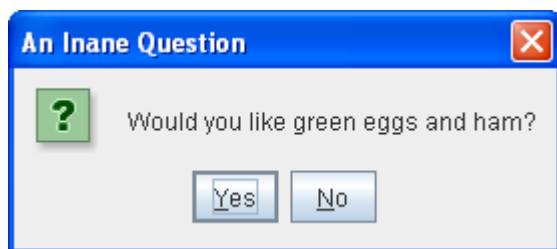
int n = JOptionPane.showOptionDialog(frame,
    "Would you like some green eggs to go "
    + "with that ham?",
    "A Silly Question",
    JOptionPane.YES_NO_CANCEL_OPTION,
    JOptionPane.QUESTION_MESSAGE,
    null,
    options,
    options[2]);

```

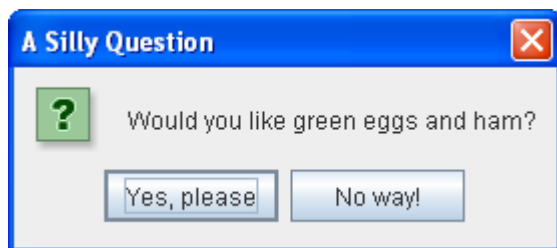
Customizing Button Text

When you use `JOptionPane` to create a dialog, you can either use the standard button text (which might vary by look and feel and locale) or specify different text. By default, the option pane type determines how many buttons appear. For example, `YES_NO_OPTION` dialogs have two buttons, and `YES_NO_CANCEL_OPTION` dialogs have three buttons.

In the following example, the first dialog is implemented with **`showConfirmDialog`**, which uses the look-and-feel wording for the two buttons. The second dialog uses **`showOptionDialog`** so it can customize the wording. With the exception of wording changes, the dialogs are identical.



```
//default icon, custom title
int n = JOptionPane.showConfirmDialog(
    frame,
    "Would you like green eggs and
    ham?",
    "An Inane Question",
    JOptionPane.YES_NO_OPTION);
```



```
Object[] options = {"Yes, please",
    "No way!"};
int n =
    JOptionPane.showOptionDialog(frame,
        "Would you like green eggs and
        ham?",
        "A Silly Question",
        JOptionPane.YES_NO_OPTION,
        JOptionPane.QUESTION_MESSAGE,
        null, //do not use a custom
        Icon
        options, //the titles of buttons
        options[0]); //default button title
```

Data Exchange in with Dialog Box

Can be achieved through event handling. (will be covered in next Unit)

Java JFileChooser

The object of **JFileChooser** class represents a dialog window from which the user can select file. It inherits JComponent class.

JFileChooser class declaration

Let's see the declaration for javax.swing.JFileChooser class.

public class JFileChooser **extends** JComponent **implements** Accessible

Commonly used Constructors:

Constructor	Description
JFileChooser()	Constructs a JFileChooser pointing to the user's default directory.
JFileChooser(File currentDirectory)	Constructs a JFileChooser using the given File as the path.
JFileChooser(String currentDirectoryPath)	Constructs a JFileChooser using the given path.

Example

```
import java.io.File;

import javax.swing.JFileChooser;
import javax.swing.filechooser.FileSystemView;

public class FileChooser1 {

    public static void main(String[] args) {

        JFileChooser jfc = new
JFileChooser(FileSystemView.getFileSystemView().getHomeDirectory());

        int returnValue = jfc.showOpenDialog(null);
        // int returnValue = jfc.showSaveDialog(null);

        if (returnValue == JFileChooser.APPROVE_OPTION) {
            File selectedFile = jfc.getSelectedFile();
            System.out.println(selectedFile.getAbsolutePath());
        }

    }

}
```

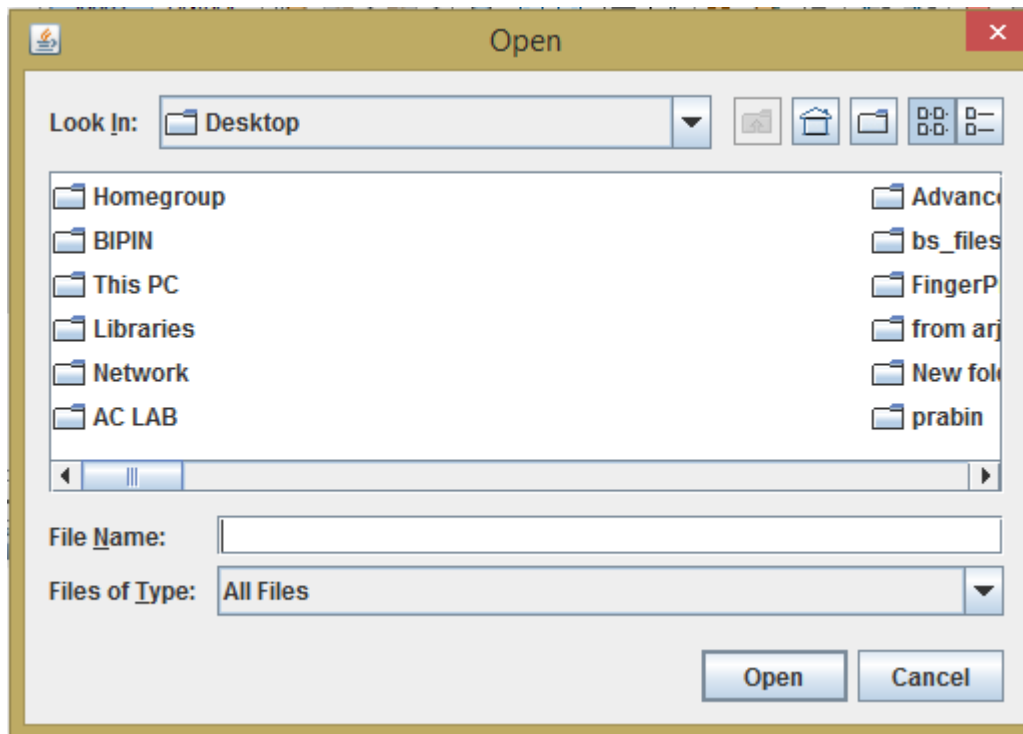



Figure 34:File Chooser Example

Java JColorChooser

The JColorChooser class is used to create a color chooser dialog box so that user can select any color. It inherits JComponent class.

Let's see the declaration for javax.swing.JColorChooser class.

public class JColorChooser **extends** JComponent **implements** Accessible

Commonly used Constructors:

Constructor	Description
JColorChooser()	It is used to create a color chooser panel with white color initially.
JColorChooser(color initialcolor)	It is used to create a color chooser panel with the specified color initially.

Commonly used Methods:

Method	Description
void addChooserPanel(AbstractColorChooserPanel panel)	It is used to add a color chooser panel to the color chooser.
static Color showDialog(Component c, String title, Color initialColor)	It is used to show the color chooser dialog box.

Java JColorChooser Example

```

1. import java.awt.event.*;
2. import java.awt.*;
3. import javax.swing.*;
4. public class ColorChooserExample extends JFrame implements ActionListener {
5.     JButton b;
6.     Container c;
7.     ColorChooserExample(){
8.         c=getContentPane();
9.         c.setLayout(new FlowLayout());
10.        b=new JButton("color");
11.        b.addActionListener(this);
12.        c.add(b);
13.    }
14.    public void actionPerformed(ActionEvent e) {
15.        Color initialcolor=Color.RED;
16.        Color color=JColorChooser.showDialog(this,"Select a color",initialcolor);
17.        c.setBackground(color);
18.    }
19.
20.    public static void main(String[] args) {
21.        ColorChooserExample ch=new ColorChooserExample();
22.        ch.setSize(400,400);
23.        ch.setVisible(true);
24.        ch.setDefaultCloseOperation(EXIT_ON_CLOSE);
25.    }
26. }

```

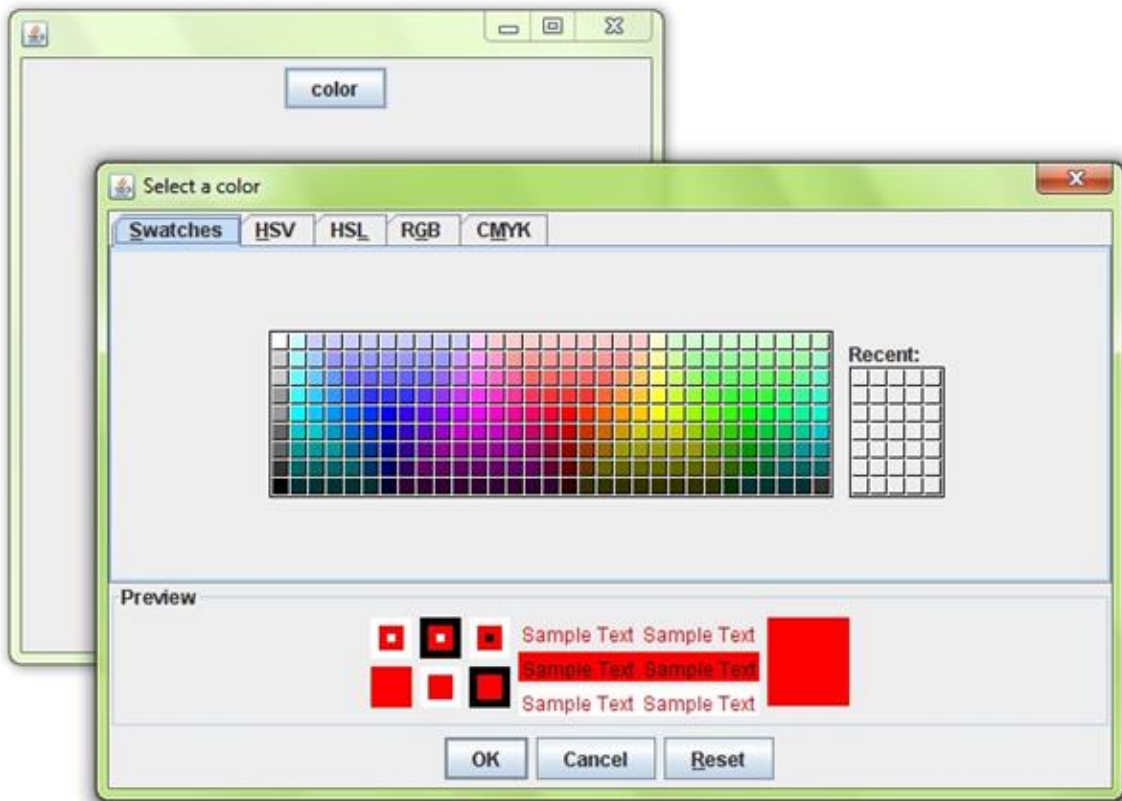


Figure 35: Color Chooser Example

Component Organizers

Java JSplitPane

JSplitPane is used to divide two components. The two components are divided based on the look and feel implementation, and they can be resized by the user. If the minimum size of the two components is greater than the size of the split pane, the divider will not allow you to resize it.

The two components in a split pane can be aligned left to right using **JSplitPane.HORIZONTAL_SPLIT**, or top to bottom using **JSplitPane.VERTICAL_SPLIT**. When the user is resizing the components the minimum size of the components is used to determine the maximum/minimum position the components can be set to.

Nested Class

Modifier and Type	Class	Description
protected class	JSplitPane.AccessibleJSplitPane	This class implements accessibility support for the JSplitPane class.

Useful Fields

Modifier and Type	Field	Description
static String	BOTTOM	It use to add a Component below the other Component.
static String	CONTINUOUS_LAYOUT_PROPERTY	Bound property name for continuousLayout.
static String	DIVIDER	It uses to add a Component that will represent the divider.
static int	HORIZONTAL_SPLIT	Horizontal split indicates the Components are split along the x axis.
protected int	lastDividerLocation	Previous location of the split pane.
protected Component	leftComponent	The left or top component.
static int	VERTICAL_SPLIT	Vertical split indicates the Components are split along the y axis.
protected Component	rightComponent	The right or bottom component.
protected int	orientation	How the views are split.

Constructors

Constructor	Description
JSplitPane()	It creates a new JSplitPane configured to arrange the child components side-by-side horizontally, using two buttons for the components.
JSplitPane(int newOrientation)	It creates a new JSplitPane configured with the specified orientation.
JSplitPane(int newOrientation, boolean newContinuousLayout)	It creates a new JSplitPane with the specified orientation and redrawing style.
JSplitPane(int newOrientation, boolean newContinuousLayout, Component newLeftComponent, Component newRightComponent)	It creates a new JSplitPane with the specified orientation and redrawing style, and with the specified components.
JSplitPane(int newOrientation, Component newLeftComponent, Component newRightComponent)	It creates a new JSplitPane with the specified orientation and the specified components.

Useful Methods

Modifier and Type	Method	Description
protected void	addImpl(Component comp, Object constraints, int index)	It adds the specified component to this split pane.
AccessibleContext	getAccessibleContext()	It gets the AccessibleContext associated with this JSplitPane.
int	getDividerLocation()	It returns the last value passed to setDividerLocation.
int	getDividerSize()	It returns the size of the divider.

Component	getBottomComponent()	It returns the component below, or to the right of the divider.
Component	getRightComponent()	It returns the component to the right (or below) the divider.
SplitPaneUI	getUI()	It returns the SplitPaneUI that is providing the current look and feel.
Boolean	isContinuousLayout()	It gets the continuousLayout property.
Boolean	isOneTouchExpandable()	It gets the oneTouchExpandable property.
Void	setOrientation(int orientation)	It gets the orientation, or how the splitter is divided.

Example

```

import java.awt.FlowLayout;
import java.awt.Panel;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JSplitPane;
public class JSplitPaneExample {
    private static void createAndShow() {
        // Create and set up the window.
        final JFrame frame = new JFrame("JSplitPane Example");
        // Display the window.
        frame.setSize(300, 300);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // set flow layout for the frame
        frame.getContentPane().setLayout(new FlowLayout());
        String[] option1 = { "A", "B", "C", "D", "E" };
        JComboBox box1 = new JComboBox(option1);
        String[] option2 = { "1", "2", "3", "4", "5" };
        JComboBox box2 = new JComboBox(option2);
    }
}

```

```

Panel panel1 = new Panel();
panel1.add(box1);
Panel panel2 = new Panel();
panel2.add(box2);
JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, panel1, pael2);
// JSplitPane splitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
// panel1, panel2);
frame.getContentPane().add(splitPane);
}
public static void main(String[] args) {
    // Schedule a job for the event-dispatching thread:
    // creating and showing this application's GUI.
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            createAndShow();
        }
    });
}
}

```

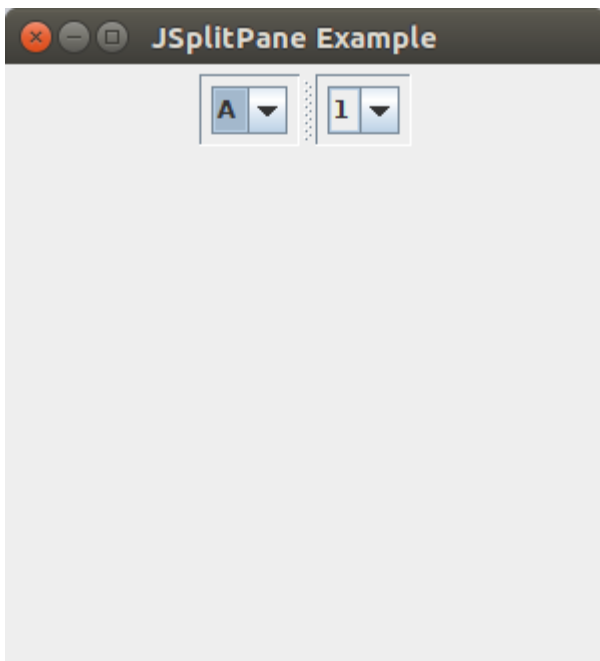


Figure 36: Split Pane Example

Java JDesktopPane

The JDesktopPane class, can be used to create "multi-document" applications. A multi-document application can have many windows included in it. We do it by making the contentPane in the main window as an instance of the JDesktopPane class or a subclass. Internal windows add instances of JInternalFrame to the JdesktopPane instance. The internal windows are the instances of JInternalFrame or its subclasses.

Fields

Modifier and Type	Field	Description
static int	LIVE_DRAG_MODE	It indicates that the entire contents of the item being dragged should appear inside the desktop pane.
static int	OUTLINE_DRAG_MODE	It indicates that an outline only of the item being dragged should appear inside the desktop pane.

Constructor

Constructor	Description
JDesktopPane()	Creates a new JDesktopPane.

Example:

```

1. import java.awt.BorderLayout;
2. import java.awt.Container;
3. import javax.swing.JDesktopPane;
4. import javax.swing.JFrame;
5. import javax.swing.JInternalFrame;
6. import javax.swing.JLabel;
7. public class JDPaneDemo extends JFrame
8. {
9.     public JDPaneDemo()
10.    {
11.        CustomDesktopPane desktopPane = new CustomDesktopPane();
12.        Container contentPane = getContentPane();
13.        contentPane.add(desktopPane, BorderLayout.CENTER);
14.        desktopPane.display(desktopPane);
15.
16.        setTitle("JDesktopPane Example");
17.        setSize(300,350);
18.        setVisible(true);
19.    }
20.    public static void main(String args[])
21.    {
22.        new JDPaneDemo();
23.    }
24. }
25. class CustomDesktopPane extends JDesktopPane
26. {
27.     int numFrames = 3, x = 30, y = 30;
28.     public void display(CustomDesktopPane dp)
29.     {
30.         for(int i = 0; i < numFrames ; ++i )
31.         {
32.             JInternalFrame jframe = new JInternalFrame("Internal Frame " + i , true, true, true, true);
33.
34.             jframe.setBounds(x, y, 250, 85);
35.             Container c1 = jframe.getContentPane( ) ;
36.             c1.add(new JLabel("I love my country"));
37.             dp.add( jframe );

```

```
38. jframe.setVisible(true);  
39. y += 85;  
40. }  
41. }  
42. }
```

Output:

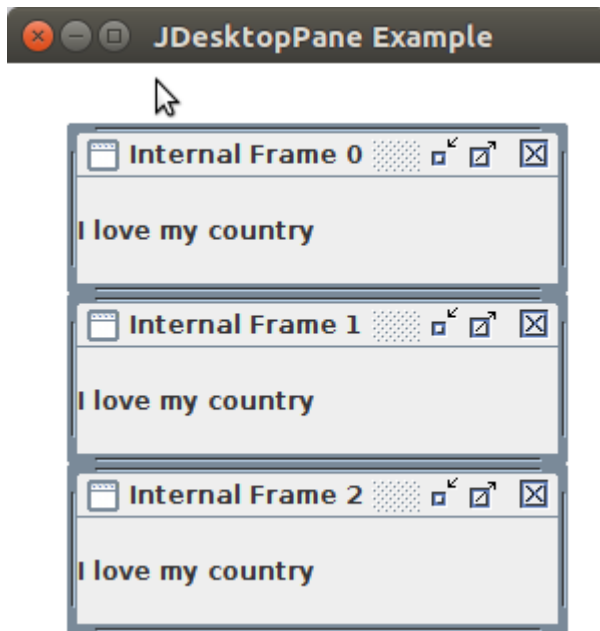


Figure 37: Desktop Pane

Java JTabbedPane

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

Let's see the declaration for javax.swing.JTabbedPane class.

public class JTabbedPane **extends** JComponent **implements** Serializable, Accessible, SwingConstants

Commonly used Constructors:

Constructor	Description
JTabbedPane()	Creates an empty TabbedPane with a default tab placement of JTabbedPane.Top.
JTabbedPane(int tabPlacement)	Creates an empty TabbedPane with a specified tab placement.
JTabbedPane(int tabPlacement, int tabLayoutPolicy)	Creates an empty TabbedPane with a specified tab placement and tab layout policy.

Example

```
import javax.swing.*;
public class TabbedPaneExample {
    JFrame f;
    TabbedPaneExample(){
        f=new JFrame();
        JTextArea ta=new JTextArea(200,200);
        JPanel p1=new JPanel();
        p1.add(ta);
        JPanel p2=new JPanel();
        JPanel p3=new JPanel();
        JTabbedPane tp=new JTabbedPane();
        tp.setBounds(50,50,200,200);
        tp.add("main",p1);
        tp.add("visit",p2);
        tp.add("help",p3);
        f.add(tp);
        f.setSize(400,400);
        f.setLayout(null);
    }
}
```

```
f.setVisible(true);  
}  
public static void main(String[] args) {  
    new TabbedPaneExample();  
}}
```

Output:

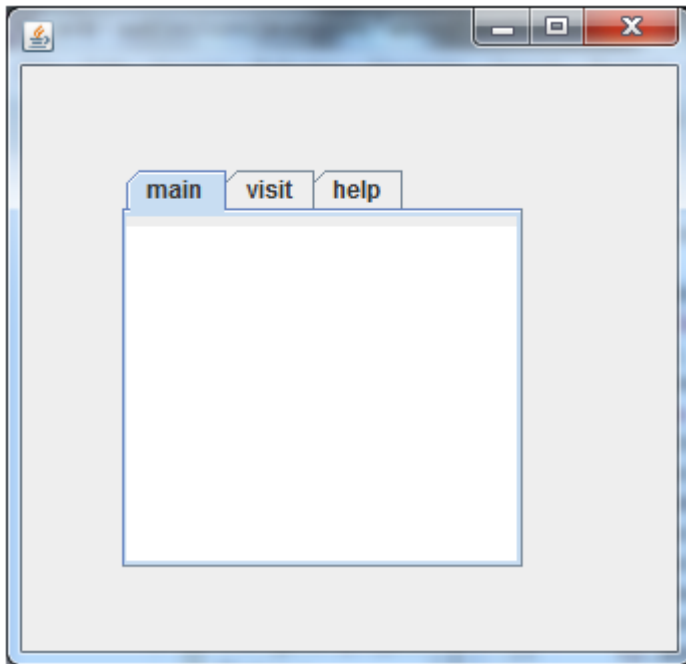


Figure 38: TabbedPane Example

JInternalFrame

With the **JInternalFrame** class you can display a **JFrame**-like window within another window. Usually, you add internal frames to a desktop pane. The desktop pane, in turn, might be used as the content pane of a **JFrame**. The desktop pane is an instance of **JDesktopPane**, which is a subclass of **JLayeredPane** that has added API for managing multiple overlapping internal frames

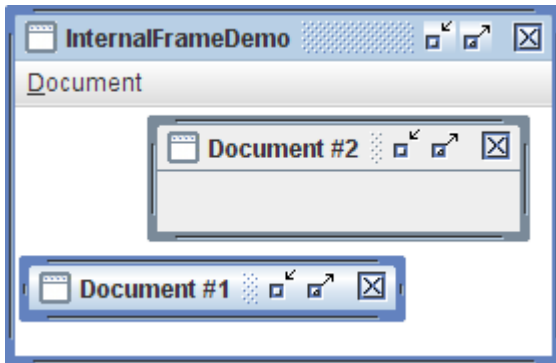


Figure 39: Internal Frame

- The code for using internal frames is similar in many ways to the code for using regular Swing frames. Because internal frames have root panes, setting up the GUI for a **JInternalFrame** is very similar to setting up the GUI for a **JFrame**. **JInternalFrame** also provides other API, such as **pack**, that makes it similar to **JFrame**.
- Just as for a regular frame, you must invoke **setVisible(true)** or **show()** on an internal frame to display it. The internal frame does not appear until you explicitly make it visible.
- Internal frames are not windows or top-level containers, however, which makes them different from frames. For example, you must add an internal frame to a container (usually a **JDesktopPane**); an internal frame cannot be the root of a containment hierarchy. Also, internal frames do not generate window events. Instead, the user actions that would cause a frame to fire window events cause an internal frame to fire internal frame events.
- Because internal frames are implemented with platform-independent code, they add some features that frames cannot give you. One such feature is that internal frames give you more control over their state and capabilities than frames do. You can programmatically iconify or maximize an internal frame. You can also specify what icon goes in the internal frame's title bar. You can even specify whether the internal frame has the window decorations to support resizing, iconifying, closing, and maximizing.
- Another feature is that internal frames are designed to work within desktop panes. The **JInternalFrame** API contains methods such as **moveToFront** that work only if the internal frame's container is a layered pane such as a **JDesktopPane**.

Some rules

➤ **You must set the size of the internal frame.**

If you do not set the size of the internal frame, it will have zero size and thus never be visible. You can set the size using one of the following methods: **setSize**, **pack**, or **setBounds**.

- **As a rule, you should set the location of the internal frame.**
If you do not set the location of the internal frame, it will come up at 0,0 (the upper left of its container). You can use the setLocation or setBounds method to specify the upper left point of the internal frame, relative to its container.
- **To add components to an internal frame, you add them to the internal frame's content pane.**
This is exactly like the JFrame situation.
- **Dialogs that are internal frames should be implemented using JOptionPane or JInternalFrame, not JDialog.**
To create a simple dialog, you can use the JOptionPane showInternalXxxDialog methods
- **You must add an internal frame to a container.**
If you do not add the internal frame to a container (usually a JDesktopPane), the internal frame will not appear.
- **You need to call show or setVisible on internal frames.**
Internal frames are invisible by default. You must invoke setVisible(true) or show() to make them visible.
- **Internal frames fire internal frame events, not window events.**
Handling internal frame events is almost identical to handling window events

Example

```
JInternalFrame jn = new JInternalFrame("InternalFrame",true,true,true);
```

The first argument specifies the title to be displayed by the internal frame. The rest of the arguments specify whether the internal frame should contain decorations allowing the user to resize, close, and maximize the internal frame . The default value for each boolean argument is false, which means that the operation is not allowed.

```
import javax.swing.JFrame;

import javax.swing.JInternalFrame;

import javax.swing.JButton;

import java.awt.FlowLayout;


class JInternalFrameTest extends JFrame {

    JInternalFrameTest()
    {
        setTitle("OuterFrame");

        setJInternalFrame();

        setSize(700,300);

        setVisible(true);

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    void setJInternalFrame()
    {
        JInternalFrame jn = new JInternalFrame("InnerFrame",true,true,true);

        jn.setSize(500,500);

        jn.setLayout(new FlowLayout());

        jn.add(new JButton("InnerBUTTON"));

        jn.setVisible(true);

        add(jn);
    }
}
```

```
}

public class JavaApp {
    public static void main(String[] args) {
        JInternalFrameTest jn = new JInternalFrameTest();
    }
}
```

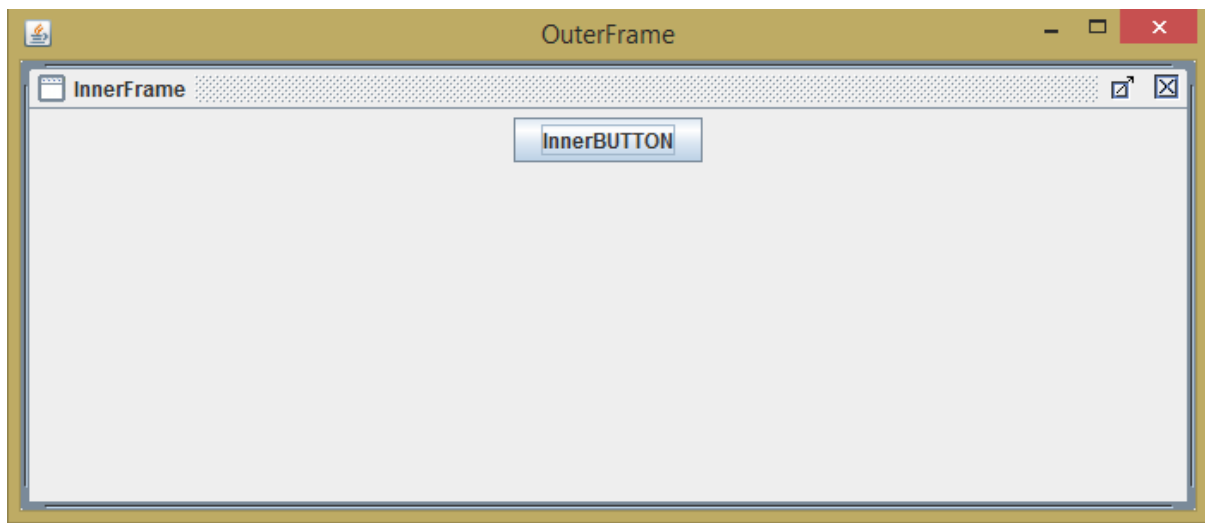


Figure 40: Internal Frame Example

Cascading and Tiling

The Java **JDesktopPane** and **JInternalFrame** classes have no built-in support for these operations. We have to make it ourselves. Cascading and Tiling is graphically shown in following figures.

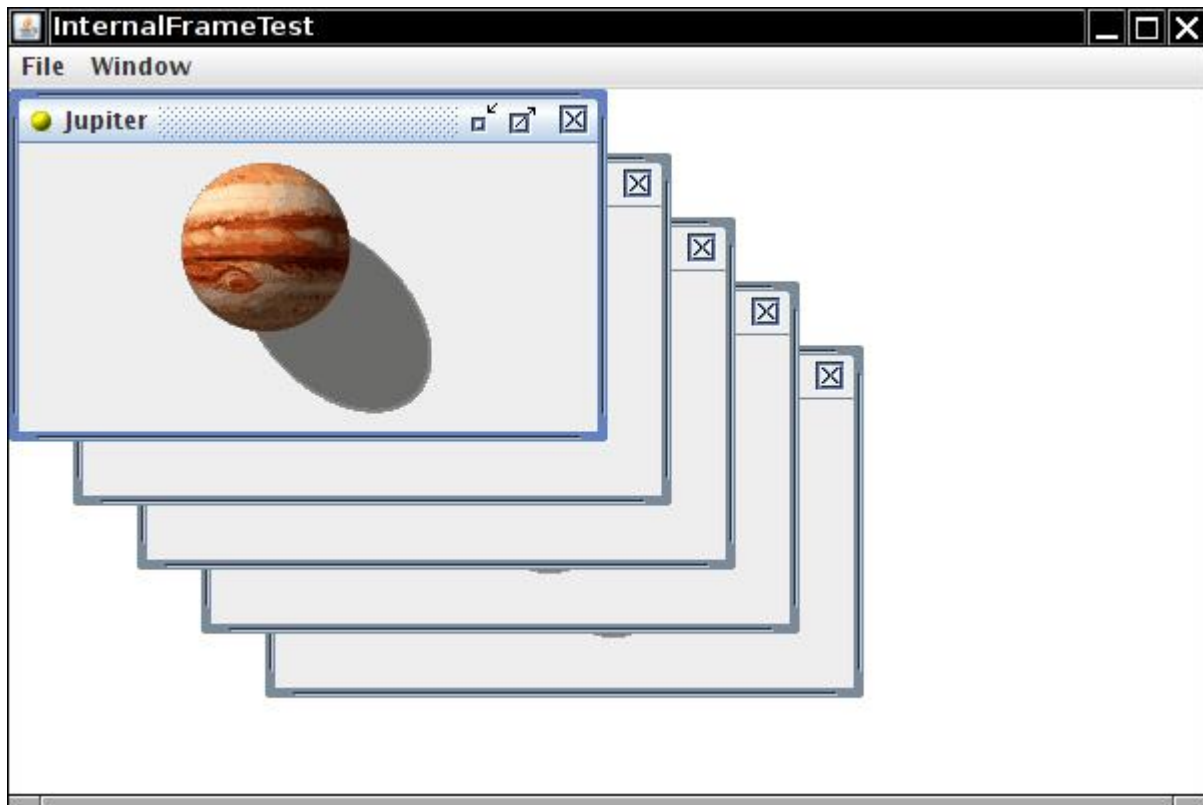


Figure 41: Cascaded Internal Frames

- ✓ To cascade all windows, you reshape windows to the same size and stagger their positions.
- ✓ The **getAllFrames()** method of the **JDesktopPane** class returns an array of all internal frames.


```
desktop = new JDesktopPane();
JInternalFrame [ ] frames = desktop.getAllFrame();
```
- ✓ An internal frame can be in one of the following three states
 - Icon
 - Resizable
 - Maximum
- ✓ You use the **isIcon()** method to find out which internal frames are currently icons and should be skipped. However, if a frame is in the maximum state, you first set it to be resizable by calling **setMaximum(false)**. This is another property that can be vetoed, so you must catch the **PropertyVetoException**.

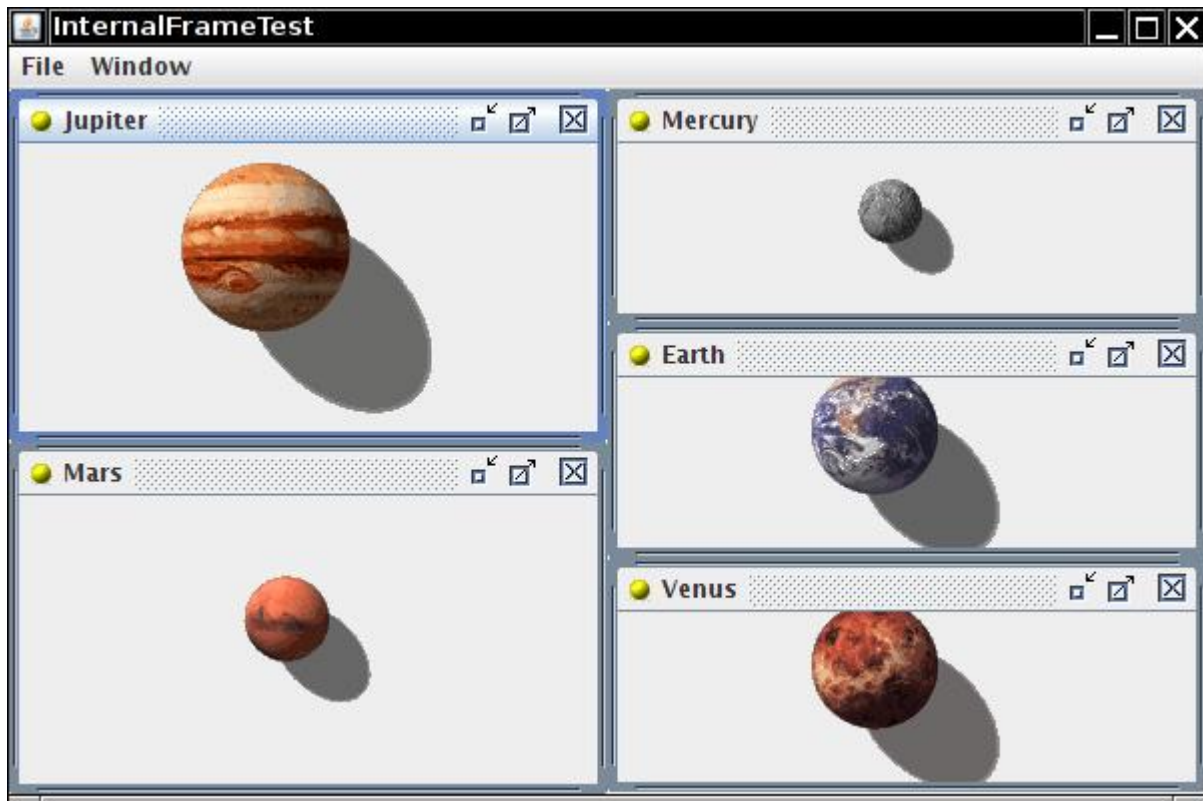


Figure 42: Tiled Internal frames

- Tiling frame is trickier, particularly if the number of frames is not a perfect square.
- First, count the number of frames that are not icons.
- Then, count the number of rows as
 - `int rows =(int) Math.sqrt(frameCount)`
- Then the number of columns is
 - `int cols = frameCount / rows;`
 except that the last
 - `int extra = frameCount % rows`
- Columns have **rows+1** rows.

(Note: Explore yourself for full codes)

Advance Swing Components

List, Tree, Table, Progress Bar

Java JList

The object of JList class represents a list of text items. The list of text items can be set up so that the user can choose either one item or multiple items. It inherits **JComponent** class.

The declaration for javax.swing.JList class.

public class JList **extends** JComponent **implements** Scrollable, Accessible

Commonly used Constructors:

Constructor	Description
JList()	Creates a JList with an empty, read-only, model.
JList(ary[] listData)	Creates a JList that displays the elements in the specified array.
JList(ListModel<ary> dataModel)	Creates a JList that displays elements from the specified, non-null, model.

Commonly used Methods:

Methods	Description
Void addListSelectionListener(ListSelectionListener listener)	It is used to add a listener to the list, to be notified each time a change to the selection occurs.
int getSelectedIndex()	It is used to return the smallest selected cell index.
ListModel getModel()	It is used to return the data model that holds a list of items displayed by the JList component.
void setListData(Object[] listData)	It is used to create a read-only ListModel from an array of objects.

Java JList Example

```
1. import javax.swing.*;
2. public class ListExample
3. {
4.     ListExample(){
5.         JFrame f= new JFrame();
6.         DefaultListModel<String> l1 = new DefaultListModel<>();
7.         l1.addElement("Item1");
8.         l1.addElement("Item2");
9.         l1.addElement("Item3");
10.        l1.addElement("Item4");
11.        JList<String> list = new JList<>(l1);
12.        list.setBounds(100,100, 75,75);
13.        f.add(list);
14.        f.setSize(400,400);
15.        f.setLayout(null);
16.        f.setVisible(true);
17.    }
18. public static void main(String args[])
19. {
20.     new ListExample();
21. }
```

Output:

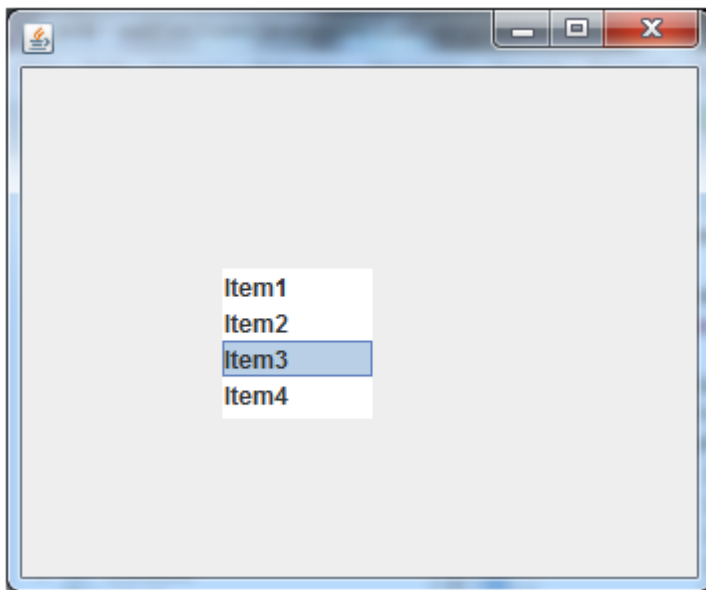


Figure 43: List Example

Java JTree

The **JTree** class is used to display the tree structured data or hierarchical data. JTree is a complex component. It has a 'root node' at the top most which is a parent for all nodes in the tree. It inherits JComponent class.

JTree class declaration

Let's see the declaration for javax.swing.JTree class.

public class JTree **extends** JComponent **implements** Scrollable, Accessible

Commonly used Constructors:

Constructor	Description
JTree()	Creates a JTree with a sample model.
JTree(Object[] value)	Creates a JTree with every element of the specified array as the child of a new root node.
JTree(TreeNode root)	Creates a JTree with the specified TreeNode as its root, which displays the root node.

Java JTree Example

```
import javax.swing.*;
import javax.swing.tree.DefaultMutableTreeNode;
public class TreeExample {
    JFrame f;
    TreeExample(){
        f=new JFrame();
        DefaultMutableTreeNode style=new DefaultMutableTreeNode("Style");
        DefaultMutableTreeNode color=new DefaultMutableTreeNode("color");
        DefaultMutableTreeNode font=new DefaultMutableTreeNode("font");
        style.add(color);
        style.add(font);
        DefaultMutableTreeNode red=new DefaultMutableTreeNode("red");
        DefaultMutableTreeNode blue=new DefaultMutableTreeNode("blue");
        DefaultMutableTreeNode black=new DefaultMutableTreeNode("black");
        DefaultMutableTreeNode green=new DefaultMutableTreeNode("green");
        color.add(red); color.add(blue); color.add(black); color.add(green);
        JTree jt=new JTree(style);
```

```

f.add(jt);
f.setSize(200,200);
f.setVisible(true);
}
public static void main(String[] args) {
    new TreeExample();
}}

```

Output:

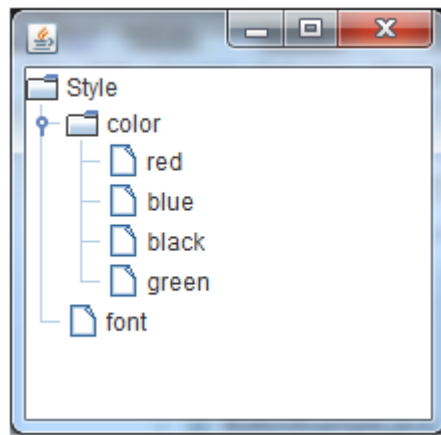


Figure 44: Tree Example

Java JTable

The JTable class is used to display data in tabular form. It is composed of rows and columns.

✓ **javax.swing.JTable** class.

Commonly used Constructors:

Constructor	Description
JTable()	Creates a table with empty cells.
JTable(Object[][] rows, Object[] columns)	Creates a table with the specified data.

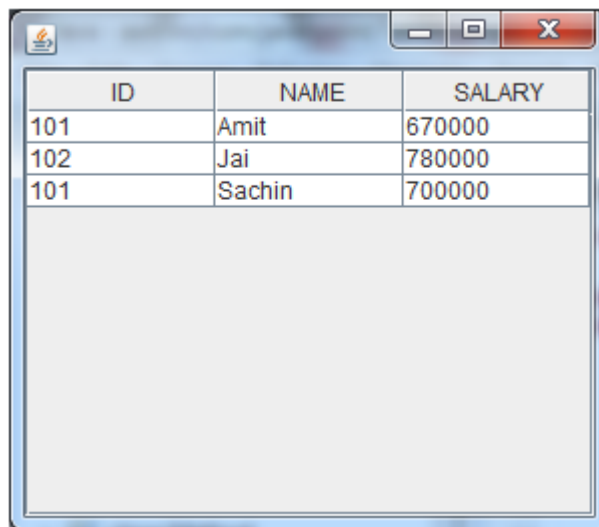
Java JTable Example

```

import javax.swing.*;

public class TableExample {
    JFrame f;
    TableExample(){
        f=new JFrame();
        String data[][]={ {"101","Amit","670000"},
                           {"102","Jai","780000"},
                           {"101","Sachin","700000"} };
        String column[]={ "ID","NAME","SALARY" };
        JTable jt=new JTable(data,column);
        jt.setBounds(30,40,200,300);
        JScrollPane sp=new JScrollPane(jt);
        f.add(sp);
        f.setSize(300,400);
        f.setVisible(true);
    }
    public static void main(String[] args) {
        new TableExample();
    }
}

```

Output:


ID	NAME	SALARY
101	Amit	670000
102	Jai	780000
101	Sachin	700000

Figure 45: Table Example

Java JProgressBar

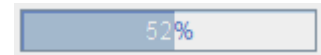


Figure 46: Progress Bar

- Sometimes a task running within a program might take a while to complete. A user-friendly program provides some indication to the user that the task is occurring, how long the task might take, and how much work has already been done. One way of indicating work, and perhaps the amount of progress, is to use an animated image
- The JProgressBar class is used to display the progress of the task. It inherits JComponent class
- The declaration for javax.swing.JProgressBar class.

public class JProgressBar **extends** JComponent **implements** SwingConstants, Accessible

Commonly used Constructors:

Constructor	Description
JProgressBar()	It is used to create a horizontal progress bar but no string text.
JProgressBar(int min, int max)	It is used to create a horizontal progress bar with the specified minimum and maximum value.
JProgressBar(int orient)	It is used to create a progress bar with the specified orientation, it can be either Vertical or Horizontal by using SwingConstants.VERTICAL and SwingConstants.HORIZONTAL constants.
JProgressBar(int orient, int min, int max)	It is used to create a progress bar with the specified orientation, minimum and maximum value.

Commonly used Methods:

Method	Description
void setStringPainted(boolean b)	It is used to determine whether string should be displayed.
void setString(String s)	It is used to set value to the progress string.
void setOrientation(int orientation)	It is used to set the orientation, it may be either vertical or horizontal by using <code>SwingConstants.VERTICAL</code> and <code>SwingConstants.HORIZONTAL</code> constants.
void setValue(int value)	It is used to set the current value on the progress bar.

Java JProgressBar Example

```

import javax.swing.*;
public class ProgressBarExample extends JFrame{
    JProgressBar jb;
    int i=0,num=0;
    ProgressBarExample(){
        jb=new JProgressBar(0,2000);
        jb.setBounds(40,40,160,30);
        jb.setValue(0);
        jb.setStringPainted(true);
        add(jb);
        setSize(250,150);
        setLayout(null);
    }
    public void iterate(){
        while(i<=2000){
            jb.setValue(i);
            i=i+20;
            try{Thread.sleep(150);}catch(Exception e){}
        }
    }
}

```

```
public static void main(String[] args) {  
    ProgressBarExample m=new ProgressBarExample();  
    m.setVisible(true);  
    m.iterate();  
}  
}
```

Output:

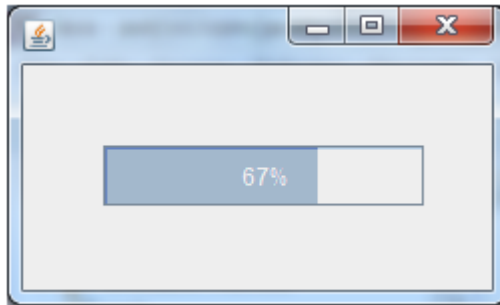


Figure 47: Progress Bar Example