ObjectZoom
We there, wherever the technology is!

2013

Advance Java (Swing)

Mr.Arvind Yadav

www.objectzoom.com

17-Jul-13

1

1.1 What Is Swing?

If you poke around the Java home page (*http://java.sun.com/*), you'll find Swing advertised as a set of customizable graphical components whose look-and-feel can be dictated at runtime. In reality, however, Swing is much more than this. Swing is the next-generation GUI toolkit that Sun Microsystems is developing to enable enterprise development in Java. By *enterprise development*, we mean that programmers can use Swing to create large-scale Java applications with a wide array of powerful components. In addition, you can easily extend or modify these components to control their appearance and behavior.

Swing is not an acronym. The name represents the collaborative choice of its designers when the project was kicked off in late 1996. Swing is actually part of a larger family of Java products known as the Java Foundation Classes (JFC), which incorporate many of the features of Netscape's Internet Foundation Classes (IFC), as well as design aspects from IBM's Taligent division and Lighthouse Design. Swing has been in active development since the beta period of the Java Development Kit (JDK)1.1, circa spring of 1997. The Swing APIs entered beta in the latter half of 1997 and their initial release was in March of 1998. When released, the Swing 1.0 libraries contained nearly 250 classes and 80 interfaces

Although Swing was developed separately from the core Java Development Kit, it does require at least JDK 1.1.5 to run. Swing builds on the event model introduced in the 1.1 series of JDKs; you cannot use the Swing libraries with the older JDK 1.0.2. In addition, you must have a Java 1.1-enabled browser to support Swing applets.

1.1.1 What Are the Java Foundation Classes (JFC)?

The Java Foundation Classes (JFC) are a suite of libraries designed to assist programmers in creating enterprise applications with Java. The Swing API is only one of five libraries that make up the JFC. The Java Foundation Classes also consist of the Abstract Window Toolkit (AWT), the Accessibility API, the 2D API, and enhanced support for drag-and-drop capabilities. While the Swing API is the primary focus of this book, here is a brief introduction to the other elements in the JFC:

AWT

The Abstract Window Toolkit is the basic GUI toolkit shipped with all versions of the Java Development Kit. While Swing does not reuse any of the older AWT components, it does build off of the lightweight component facilities introduced in AWT 1.1.

Accessibility

The accessibility package provides assistance to users who have trouble with traditional user interfaces. Accessibility tools can be used in conjunction with devices such as audible text readers or braille keyboards to allow direct access to the Swing components. Accessibility is split into two parts: the Accessibility API, which is shipped with the Swing distribution, and the Accessibility Utilities API, distributed separately.

2D *API*

The 2D API contains classes for implementing various painting styles, complex shapes, fonts, and colors. This Java package is loosely based on APIs that were licensed from IBM's

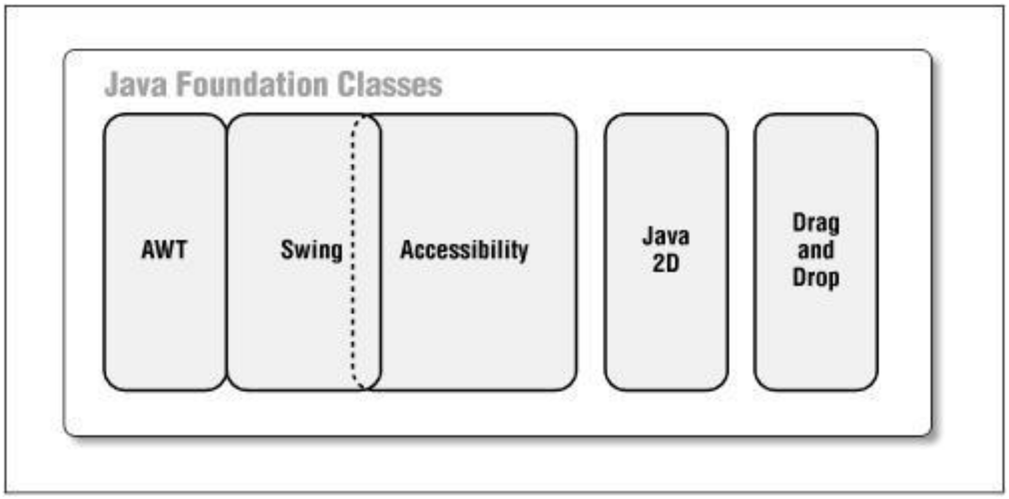
Taligent division. The 2D API classes are not part of Swing, so they will not be covered in this book.

Drag and Drop

Drag and drop is one of the more common metaphors used in graphical interfaces today. The user is allowed to click and "hold" a GUI object, moving it to another window or frame in the desktop with predictable results. The Drag and Drop API allows users to implement droppable elements that transfer information between Java applications and native applications. Drag and Drop is also not part of Swing, so we will not discuss it here.

Figure 1.1 enumerates the various components of the Java Foundation Classes. Because part of the Accessibility API is shipped with the Swing distribution, we show it overlapping Swing.

Figure 1.1. The five APIs of the Java Foundation Classes



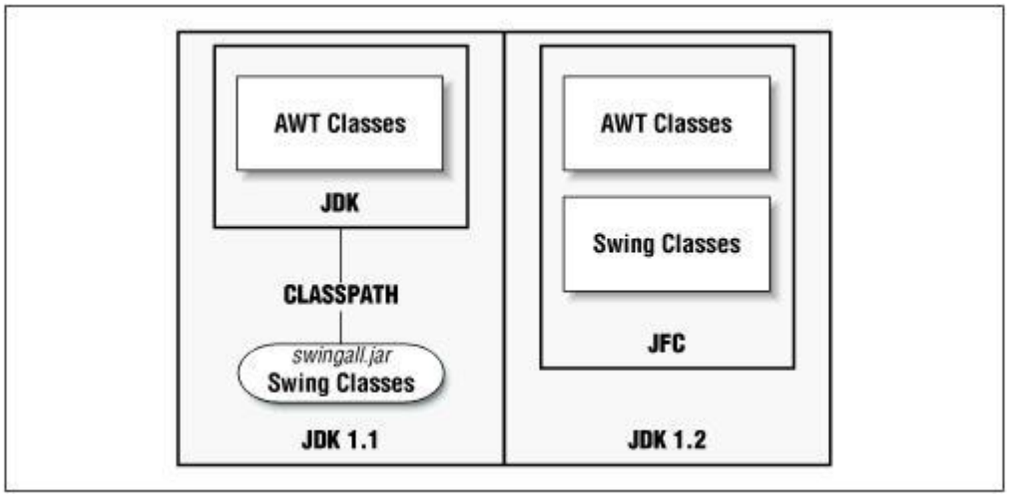
1.1.2 Is Swing a Replacement for AWT?

No. Swing is actually built on top of the core 1.1 and 1.2 AWT libraries. Because Swing does not contain any platform-specific (native) code, you can deploy the Swing distribution on any platform that implements the Java 1.1.5 virtual machine or above. In fact, if you have JDK 1.2 on your platform, then the Swing classes will already be available and there's nothing further to download. If you do not have JDK 1.2, you can download the entire set of Swing libraries as a set of Java Archive (JAR) files from the Swing home page: *http://java.sun.com/products/jfc* . In either case, it is generally a good idea to visit this URL for any extra packages or look-and-feels that may be distributed separately from the core Swing libraries.

Figure 1.2 shows the relationship between Swing, AWT, and the Java Development Kit in both the 1.1 and 1.2 JDKs. In JDK 1.1, the Swing classes must be downloaded separately and included as an archive file on the classpath (*swingall.jar*).[1] JDK 1.2 comes with a Swing distribution, although therelationship between Swing and the rest of the JDK has shifted during the beta process.

Nevertheless, if you have installed JDK 1.2, you should have Swing.

Figure 1.2. Relationships between Swing, AWT, and the JDK in the 1.1 and 1.2 JDKs



Swing contains nearly twice the number of graphical components as its immediate predecessor, AWT 1.1. Many are components that have been scribbled on programmer wish-lists since Java first debuted—including tables, trees, internal frames, and a plethora of advanced text components. In addition, Swing contains many design advances over AWT. For example, Swing introduces a new Action class that makes it easier to coordinate GUI components with the functionality they perform. You'll also find that a much cleaner design prevails throughout Swing; this cuts down on the number of unexpected surprises that you're likely to face while coding

Swing depends extensively on the event handling mechanism of AWT 1.1, although it does not define a comparatively large amount of events for itself. Each Swing component also contains a variable number of exportable properties. This combination of properties and events in the design was no accident. Each of the Swing components, like the AWT 1.1 components before them, adhere to the popular JavaBeans specification. As you might have guessed, this means that you can import all of the Swing components into various GUI-builder tools—useful for powerful visual programming.

Major difference between swing and AWT component

Sr.No.	AWT	swing
1	Heavy weight	Light weight
2	Native component	Pure java component
3	Native look and feel	Better look and feel
4	Does not have a complex component	Has additional components like JTree , JTable ,JProgressBar ,JSlider , etc
5	Applet can not have menu	JApplet have menu
6	List has scrollbar	JList does not support scrolling but this can be done using scrollPane
7	Components can be added directly on the Frame or window	While adding components on Frame or window , they have to be added on it's content pane.
8	Does not have slidePane or TabbedPane	Has alidePane or TabbedPane
9	Does not support MDI window	MDI can be achieved using InternalFrame Object
10	Menu item cannot have image or radio button or checkboxes.	Menu item can have images or radio buttons or checkboxes.
11	They do not have JMV (java Model Voewport)	All swng components have JMV
12	Default layout is flowLayout	Default layout is BorderLayout

1.2 Swing Packages and Classes

1.2.1 Swing Packages

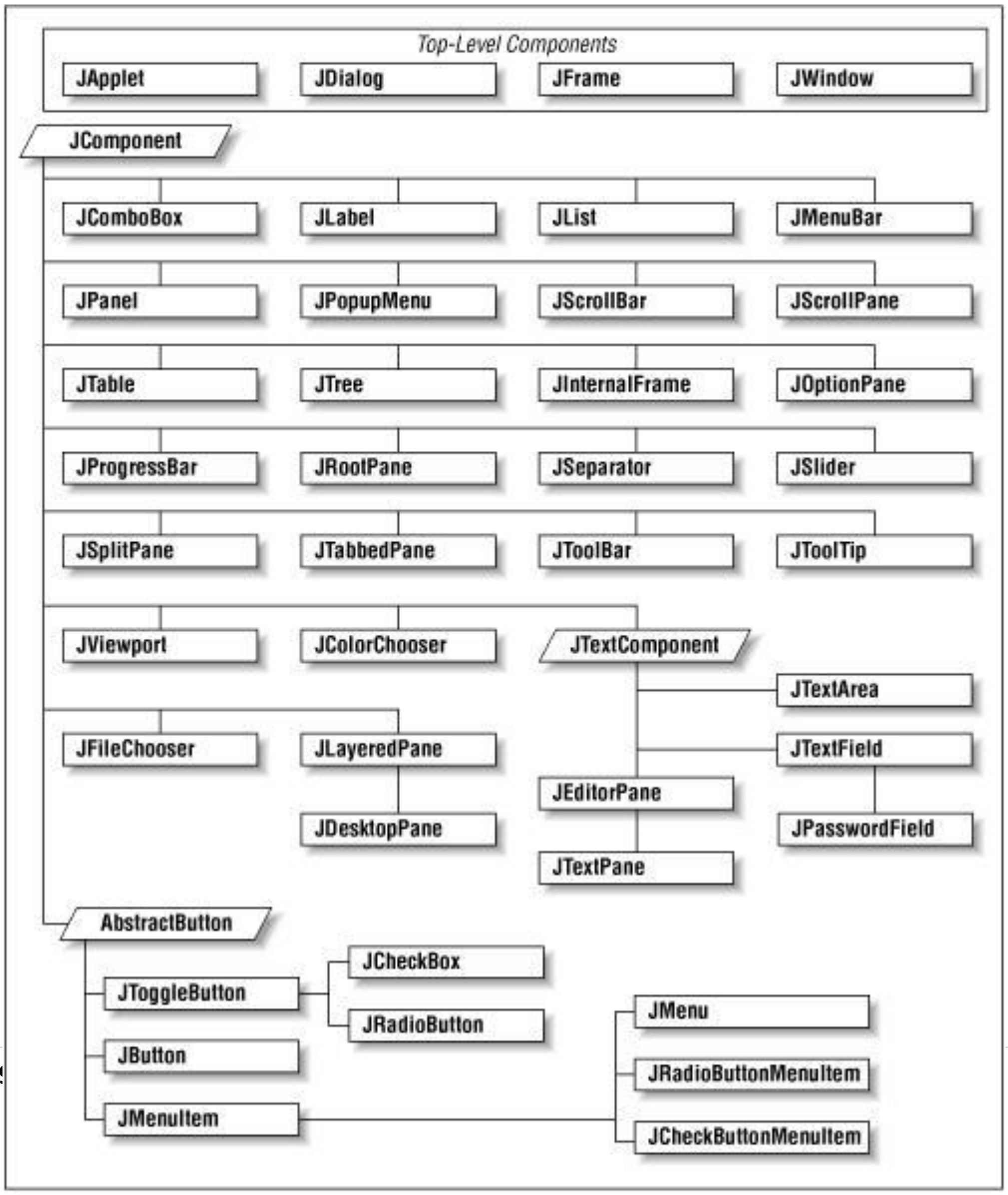
javax.swing
Contains the core Swing components, including most of the model interfaces and support

classes.

1.2.2 Class Hierarchy

Figure 1.4 shows a detailed overview of the Swing class hierarchy as it appears in the 1.2 JDK. At first glance, the class hierarchy looks very similar to AWT. Each Swing component with an AWT equivalent shares the same name, except that the Swing class is preceded by a capital "J". In most cases, if a Swing component supersedes an AWT component, it can be used as a drop-in replacement.

Figure 1.4. The Swing component hierarchy



SWING COMPONENT

Swing is a set of classes that provides more powerful and flexible components than are possible with the AWT.

Swing provide a set of “lightweight” (all-java language) component that to the maxim degree possible, Work the same on all platforms.

The term lightweight is used to describe such elements. The swing component class that are used to shown here .

Class	Description
AbstractButton	Abstract superclass for Swing buttons.
ButtonGroup	Encapsulates a mutually exclusive set of buttons.
ImageIcon	Encapsulates an icon.
JApplet	The Swing version of **Applet**.
JButton	The Swing push button class.
JCheckBox	The Swing check box class.
JComboBox	Encapsulates a combo box (an combination of a drop-down list and text field).
JLabel	The Swing version of a label.
JRadioButton	The Swing version of a radio button.
JScrollPane	Encapsulates a scrollable window.
JTabbedPane	Encapsulates a tabbed window.
JTable	Encapsulates a table-based control.
JTextField	The Swing version of a text field.

JTree	Encapsulates a tree-based control

THE JCOMPONENT CLASS

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Container └
          javax.swing.JComponet
```

With the exception of top-level containers, all Swing components whose names begin with "J" descend from the`JComponent` class. For example, `JPanel`, `JScrollPane`,`JButton`, and `JTable` all inherit from `JComponent`. However, `JFrame` and `JDialog` don't because they implement top-level containers.

The `JComponent` class extends the `Container` class, which itself extends `Component`. The `Component` class includes everything from providing layout hints to supporting painting and events. The `Container` class has support for adding components to the container and laying them out. This section's API tables summarize the most often used methods of `Component` and `Container`, as well as of`JComponent`.

JComponent Features

The **`JComponent`** class provides the following functionality to its descendants:

- Tool tips
- Painting and borders
- Application-wide pluggable look and feel
- Custom properties
- Support for layout
- Support for accessibility
- Support for drag and drop
- Double buffering
- Key bindings

Tool tips

By specifying a string with the `setToolTipText`method, you can provide help to users of a component. When the cursor pauses over the component, the specified string is displayed in a small window that appears near the component

Painting and borders

The `setBorder` method allows you to specify the border that a component displays around its edges. To paint the inside of a component, override the`paintComponent` method.

Application-wide pluggable look and feel

Behind the scenes, each `JComponent` object has a corresponding `ComponentUI` object that performs all the drawing, event handling, size determination, and so on for that `JComponent`. Exactly which`ComponentUI` object is used depends on the current look and feel, which you can set using the`UIManager.setLookAndFeel` method

Custom properties

You can associate one or more properties (name/object pairs) with any `JComponent`. For example, a layout manager might use properties to associate a constraints object with each `JComponent`it manages. You put and get properties using the`putClientProperty` and `getClientProperty`methods.

Support for layout

Although the `Component` class provides layout hint methods such as `getPreferredSize` and`getAlignmentX`, it doesn't provide any way to set these layout hints, short of creating a subclass and overriding the methods. To give you another way to set layout hints, the `JComponent` class adds setter methods — `setMinimumSize`, `setMaximumSize`,`setAlignmentX`, and `setAlignmentY`

Support for accessibility

The `JComponent` class provides API and basic functionality to help assistive technologies such as screen readers get information from Swing components.

Support for drag and drop

The `JComponent` class provides API to set a component's transfer handler, which is the basis for Swing's drag and drop support.

Double buffering

Double buffering smooths on-screen painting.

Key bindings

This feature makes components react when the user presses a key on the keyboard. For example, in many look and feels when a button has the focus, typing the Space key is equivalent to a mouse click on the button. The look and feel automatically sets up the bindings between pressing and releasing the Space key and the resulting effects on the button.

USING TOP-LEVEL CONTAINERS

1) JApplet

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Container
          └ java.awt.Panel
              └ java.applet.Applet └
                  javax.swing.JApplt
```

An extended version of java.applet.Applet that adds support for the JFC/Swing component architecture. The JApplet class is slightly incompatible with java.applet.Applet. JApplet contains a JRootPane as it's only child.
The **contentPane** should be the parent of any children of the JApplet.

to add the child to the JApplet's contentPane we use the getContentPane() method and add the components to the contentpane.

The same is true for setting LayoutManagers, removing components, listing children, etc. All these methods should normally be sent to the contentPane() instead of the JApplet itself. The contentPane() will always be non-null. Attempting to set it to null will cause the JApplet to throw an exception. The default contentPane() will have a BorderLayout manager set on it.

Constructor
JApplet()   Creates a swing applet instance.

Methods

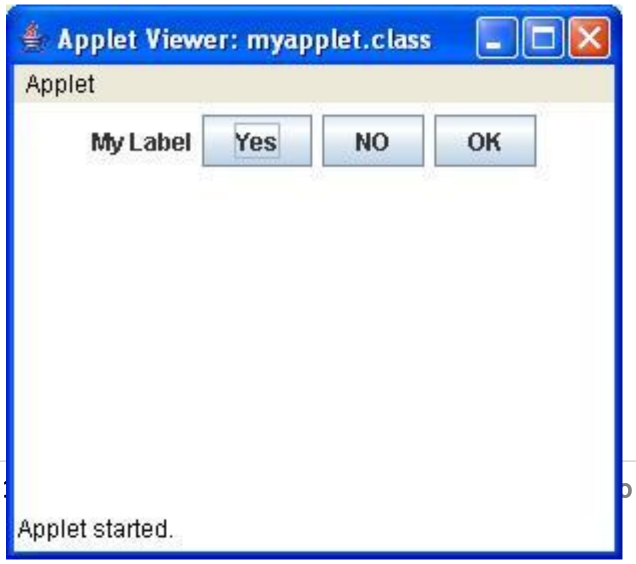
Container	**getContentPane**() Returns the contentPane object for this applet.
void	**setJMenuBar**(**JMenuBar** menuBar) Sets the menubar for this applet.
void	**setLayout**(**LayoutManager** manager) By default the layout of this component may not be set,

	the layout of its contentPane should be set instead.
void	**remove(Component comp)** Removes the specified component from this container.

Example: 01

```
import javax.swing.*;
import java.awt.*;
/*<applet code="myapplet.class" height=200
width=300></applet>*/ public class myapplet extends JApplet
{
public void init()
{
JButton btn1,btn2,btn3;
JLabel lbl;
Container cp=getContentPane();
cp.setBackground(Color.white);
cp.setLayout(new FlowLayout());
btn1=new JButton("Yes");
btn2=new JButton("NO");
btn3=new JButton("OK");
lbl=new JLabel("My Label");
cp.add(lbl);
cp.add(btn1);
cp.add(btn2);
cp.add(btn3);
}
}
```

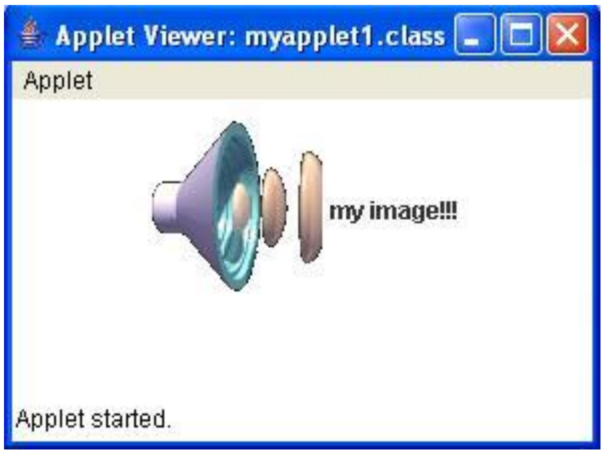
OUTPUT



Example : 02

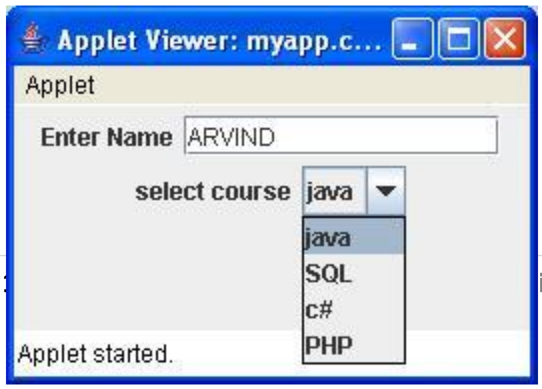
```
import java.awt.*;
import javax.swing.*;

/*<applet code="myapplet1.class" width=550 height=500></applet>*/

public class myapplet1 extends JApplet
{

        public void init()
        {
          Container cp=getContentPane();
          cp.setLayout(new FlowLayout());
          cp.setBackground(Color.white);
          ImageIcon ii=new ImageIcon("sound.gif");
          JLabel lbl=new JLabel("my image!!!",ii,JLabel.CENTER);
          cp.add(lbl);
        }
}
```

OUTPUT



Example : 03

```
import java.awt.*;
import javax.swing.*;
/*
  <applet code="myapp.class" width=300
  height=50> </applet>
*/
public class myapp extends JApplet
{
  JTextField jtf;
  JLabel lblname,lblcrs;
  JComboBox jc; public
  void init()
  {

   Container cp = getContentPane();
   cp.setLayout(new FlowLayout());

  lblname =new JLabel("Enter Name");
  lblcrs =new JLabel("select course");
    jtf = new JTextField(14);
     jc = new JComboBox();

jc.addItem("java");
jc.addItem("SQL");
jc.addItem("c#");
jc.addItem("PHP");

  cp.add(lblname);
  cp.add(jtf);
  cp.add(lblcrs);
  cp.add(jc);

}
}
```


Applet Viewer: myapp.c...

Applet

Enter Name ARVIND

select course java

java
SQL
c#
PHP

Applet started.

2) JDialog

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Container
          └ java.awt.Window
              └ java.awt.Dialog
                  └ javax.swing.JDialog
```

The JDialog is the main class for creating a dialog window. You can use this class to create a custom dialog, or invoke the many class methods in **JOptionPane** to create a variety of standard dialogs.

The `JDialog` component contains a `JRootPane` as its only child. The `contentPane` should be the parent of any children of the `JDialog`.

As a conveniance `add` andvariants, `remove` and `setLayout`have been overridden to forward to the `contentPane` as necessary

You can add an element in dialog as follows:

dialog.add(child);

Constructor Summary
JDialog()   Creates a non-modal dialog without a title and without a specified `Frame` owner.
JDialog(Dialog owner)   Creates a non-modal dialog without a title with the specified `Dialog` as its owner.
JDialog(Dialog owner, boolean modal)   Creates a modal or non-modal dialog without a title and with the specified owner dialog.
JDialog(Dialog owner, String title)   Creates a non-modal dialog with the specified title and with the specified owner dialog.

`JDialog``(Dialog owner, String title, boolean modal)`

Creates a modal or non-modal dialog with the specified title and the specified owner frame.

JDialog(Dialog owner, String title, boolean modal, GraphicsConfiguration gc) Creates a modal or non-modal dialog with the specified title, owner `Dialog`, and`GraphicsConfiguration`.
JDialog(Frame owner) Creates a non-modal dialog without a title with the specified `Frame` as its owner.
JDialog(Frame owner, boolean modal) Creates a modal or non-modal dialog without a title and with the specified owner `Frame`.
JDialog(Frame owner, String title) Creates a non-modal dialog with the specified title and with the specified owner frame.
JDialog(Frame owner, String title, boolean modal) Creates a modal or non-modal dialog with the specified title and the specified owner `Frame`.

Method Summary

protected void	**addImpl**(Component comp, Object constraints, int index) Adds the specified child `Component`.
void	**remove(Component comp)** **Removes the specified component from the container.**

3) JWindow

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Container
          └ java.awt.Window └
              javax.swing.JWindw
```

A `JWindow` is a container that can be displayed anywhere on the user's desktop. **It does not have the title bar,** window-management buttons, or other trimmings associated with a `JFrame`, but it is still a "first-class citizen" of the user's desktop, and can exist anywhere on it.

The `JWindow` component contains a `JRootPane` as its only child. The `contentPane` should be the parent of any children of the `JWindow`.

```
You can add an element in dialog as follows:
window.add(child);
```

However, using `JWindow` you would code:

```
        window.getContentPane().add(child);
```

The same is true of setting `LayoutManager`s, removing components, listing children, etc. All these methods should normally be sent to the `contentPane` instead of the `JWindow` itself. The `contentPane`will always be non-`null`. Attempting to set it to `null` will cause the `JWindow` to throw an exception. The default `contentPane` will have a `BorderLayout` manager set on it.

Constructor Summary
JWindow()   Creates a window with no specified owner.
JWindow(Frame owner)

Creates a window with the specified owner frame.

`JWindow``(GraphicsConfiguration gc)`

Creates a window with the specified `GraphicsConfiguration` of a screen device.

JWindow(Window owner) Creates a window with the specified owner window.
JWindow(Window owner, GraphicsConfiguration gc) Creates a window with the specified owner window and `GraphicsConfiguration` of a screen device.

Method Summary

Container	**getContentPane**() Returns the `Container` which is the `contentPane` for this window.
void	**setLayout**(**LayoutManager** manager) **By default the layout of this component may not be set, the layout of its contentPane should be set instead.**
void	**update**(**Graphics** g) **Calls paint(g).**
protected void	**windowInit**() **Called by the constructors to init the JWindow properly.**

4) JFrame

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Container
          └ java.awt.Window
              └ java.awt.Frame  └
                  javax.swing.JFrae
```

→ A Frame is a top-level window with a title and a border.

→ JFrame is a subclass of JWindow that has a border and can hold a menubar.You can drag a form around on the screen and resize it, using the ordinary controls for your windowing environment.

→ All other swing components and containers must be held, at Same level, inside a frame.

→ JFrame are the only components that can be displayed without being added or attached to another Container.

After creating a JFrame, you can call
setVisible()method To display it.

Creating a JFrame Window:

Step1: Construct an object of the JFrame
class Step2: Set the size of the JFrame.
Step3: Set the title of the JFrame to appear in the title (Title bar will be blank if no title is set).
Step4: Set the default close operation. When the user clicks the Close button, the program stops running.
Step5: Make the JFrame visible.

Constructors
JFrame()   Constructs a new frame that is initially invisible.
JFrame(GraphicsConfiguration gc)   Creates a `Frame` in the specified `GraphicsConfiguration` of a screen device and a blank title.
JFrame(String title)   Creates a new, initially invisible `Frame` with the specified title.
JFrame(String title, GraphicsConfiguration gc)   Creates a `JFrame` with the specified title and the specified `GraphicsConfiguration` of a screen device.

Methods	
void	**setDefaultCloseOperation**(int operation) Sets the operation that will happen by default when the user initiates a "close" on this frame.
void	**setLayout**(**LayoutManager** manager) By default the layout of this component may not be set, the layout of itscontentPane should be set instead.
void	**setJMenuBar**(**JMenuBar** menubar) Sets the menubar for this frame.
void	**remove**(**Component** comp) Removes the specified component from this container.

void	**pack()** Causes this Window to be sized to fit the preferred size and layouts of its subcomponents.

`public void setDefaultCloseOperation(int operation)`

Sets the operation that will happen by default when the user initiates a "close" on this frame. You must specify one of the following choices:

- `DO_NOTHING_ON_CLOSE` (defined in `WindowConstants`): Don't do anything; require the program to handle the operation in the `windowClosing` method of a registered`WindowListener` object.
- `HIDE_ON_CLOSE` (defined in `WindowConstants`): Automatically hide the frame after invoking any registered `WindowListener` objects.
- `DISPOSE_ON_CLOSE` (defined in `WindowConstants`): Automatically hide and dispose the frame after invoking any registered `WindowListener` objects.
- `EXIT_ON_CLOSE` (defined in `JFrame`): Exit the application using the `System exit` method. Use this only in applications.

The value is set to `HIDE_ON_CLOSE` by default.

Parameters:

`operation` - the operation which should be performed when the user closes the frame

How to create a frame in Java Swing Application

The frame in java works like the main window where your components (controls) are added to develop an application. In the Java Swing, top-level windows are represented by the **JFrame** class. Java supports the look and feel and decoration for the frame.

For creating java standalone application you must provide GUI for a user. The most common way of creating a frame is, using single argument constructor of the **JFrame** class. The argument of the constructor is the title of the window or frame. Other user interface are added by constructing and adding it to the container one by one. The frame initially are not visible and to make it visible the setVisible(true) function is called passing the boolean value *true*. The close button of the frame by default performs the hide operation for the JFrame. In this example we have changed this behavior to window close operation by setting the setDefaultCloseOperation() to EXIT_ON_CLOSE value.

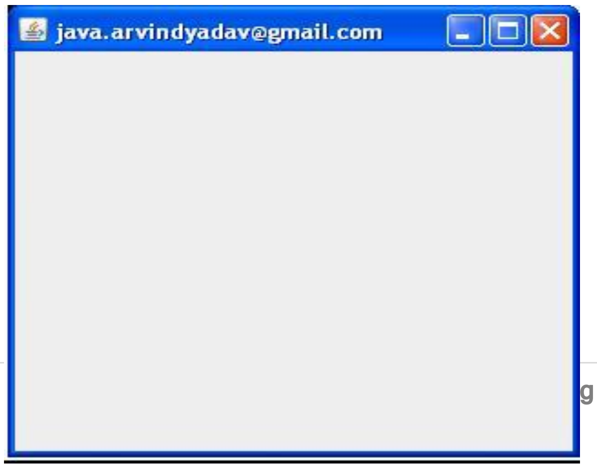
setSize (400, 400):
Above method sets the size of the frame or window to width (400) and height (400) pixels.

setVisible(true):
Above method makes the window visible.

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE):
Above code sets the operation of close operation to Exit the application using the System exit method.

Example 04:

```
import javax.swing.*;

public class Swing_Create_Frame{
  public static void main(String[] args){
    JFrame frame = new JFrame("arvind@objectzoom.com");
    frame.setSize(400, 400);
    frame.setVisible(true);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
}
```



how to make a frame non resizable. It means, disabling the maximize button of the frame.

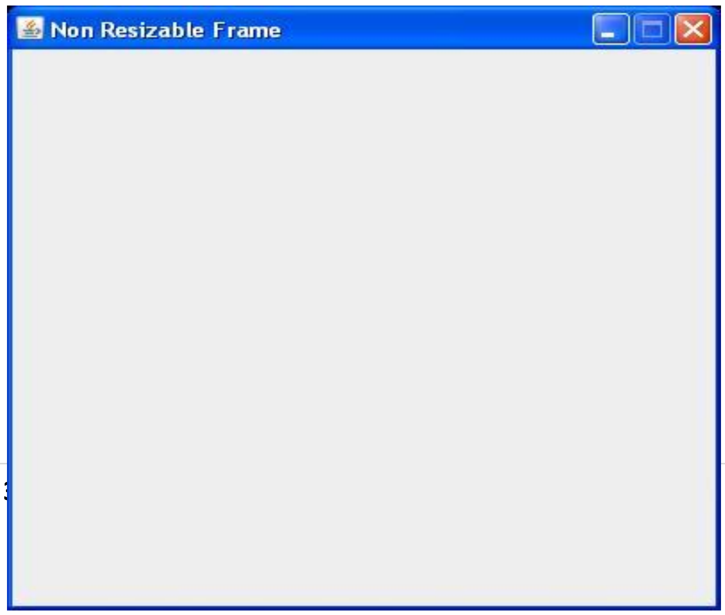
The setResizable() method has been used to make the frame resizable or not. If you pass the boolean value *false* to the setResizable() method then the frame will be non-resizable otherwise frame will be resizable. The setResizable() is the method of the **JFrame** class which takes a boolean valued argument (*true* or *false*).

Screen shot for the result of the program:

Example 05 :

```
import javax.swing.*;

public class SwingFrameNonResizable

{
   public static void main(String[] args)

    {
     JFrame frame = new JFrame("Non Resizable Frame");
     frame.setResizable(false);
     frame.setSize(400, 400);
     frame.setVisible(true);
     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
}
```

OUTPUT:



How to set an icon for the frame in Java Swing.

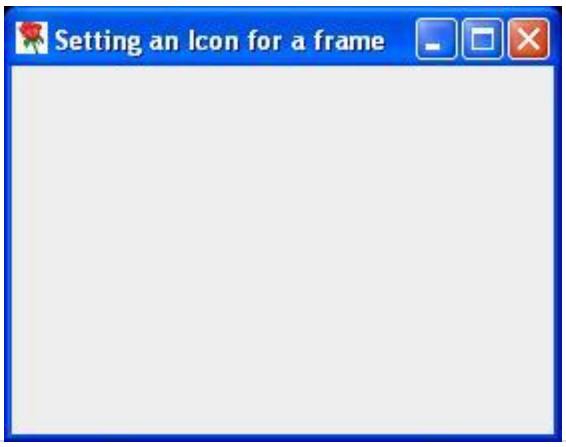
This program helps us to set the icon (image) on the title bar of the frame. When you open frame or window the icon situated on the title bar is seen on the taskbar also. For this purposes, various methods as follows has been used:

frame.setIconImage(Toolkit.getDefaultToolkit().getImage("icon_confused.gif"));
Above method sets the icon for the frame or window after getting the image using the **Image** class method named getImage().

frame.getDefaultToolkit():
This is the method of the **Toolkit** class which gets the default toolkit.

Example 06:

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

public class SettingIconFrame
{
  public static void main(String[] args)
{
  JFrame frame = new JFrame("Setting an Icon for a frame");
  frame.setIconImage(Toolkit.getDefaultToolkit().getImage("rose.gif"));
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setSize(400,400);
  frame.setVisible(true);
 }
}
```



2

USING INTERMEDIATE SWING CONTAINER S

1) Class JPanel

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.JPanel
```

The JPanel class provides general-purpose containers for lightweight components. By default, panels do not add colors to anything except their own background; however, you can easily add borders to them and otherwise customize their painting.

In many types of look and feel, panels are opaque by default. Opaque panels work well as content panes and can help with painting efficiently, as described in Using Top-Level Containers. You can change a panel's transparency by invoking the setOpaque method

Adding Components

When you add components to a panel, you use the `add`method. Exactly which arguments you specify to the `add`method depend on which layout manager the panel uses. When the layout manager is `FlowLayout`, `BoxLayout`,`GridLayout`, or `SpringLayout`, you will typically use the one-argument `add` method, like this:

```
aFlowPanel.add(aComponent);
aFlowPanel.add(anotherComponent);
```

When the layout manager is `BorderLayout`, you need to provide an argument specifying the added component's position within the panel. For example:

```
aBorderPanel.add(aComponent, BorderLayout.CENTER);
aBorderPanel.add(anotherComponent, BorderLayout.SOUTH);
```

Constructor	Purpose

JPanel()
JPanel(LayoutManager)
	Creates a panel. The`LayoutManager` parameter provides a layout manager for the new panel. By default, a panel uses a `FlowLayout` to lay out its components.

Method	Purpose
void add(Component) void add(Component, int) void add(Component, Object) void add(Component, Object, int) void add(String, Component)	Adds the specified component to the panel. When present, the `int`parameter is the index of the component within the container. By default, the first component added is at index 0, the second is at index 1, and so on. The`Object` parameter is layout manager dependent and typically provides information to the layout manager regarding positioning and other layout constraints for the added component. The `String`parameter is similar to the`Object` parameter.
void remove(Component) void remove(int) void removeAll()	Removes the specified component(s).
void setLayout(LayoutManager) LayoutManager getLayout()	Sets or gets the layout manager for this panel. The layout manager is responsible for positioning the panel's components within the panel's bounds according to some philosophy.

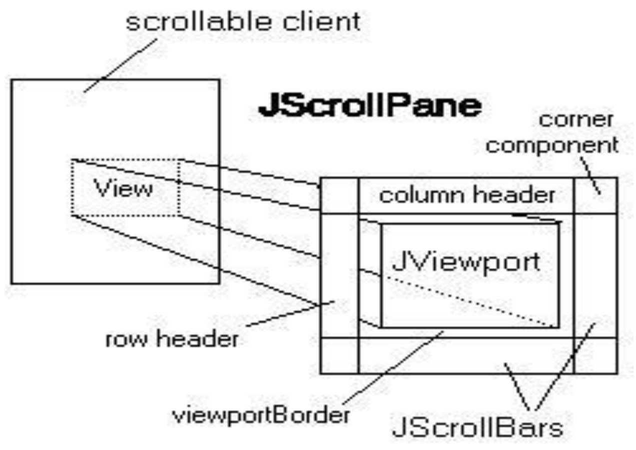
2) Class JScrollPane

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Container
          └ javax.swing.JComponent └
              javax.swing.JScrollPane
```

A JScrollPane provides a scrollable view of a component.

A JScrollPane manages a viewport, optional vertical and horizontal scroll bars, and optional row and column heading viewports.

The `JViewport` provides a window, or "viewport" onto a data source -- for example, a text file. That data source is the "scrollable client" (aka data model) displayed by the`JViewport` view. A `JScrollPane` basically consists of`JScrollBar`s, a `JViewport`, and the wiring between them, as shown in the diagram at right.



In addition to the scroll bars and viewport, a `JScrollPane`can have a column header and a row header. Each of these is a `JViewport` object that you specify with`setRowHeaderView`, and `setColumnHeaderView`.

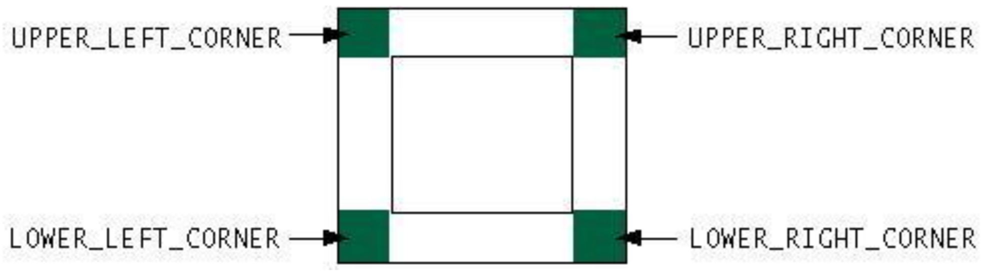
To add a border around the main `scrollPane.getViewport().setBackground()`

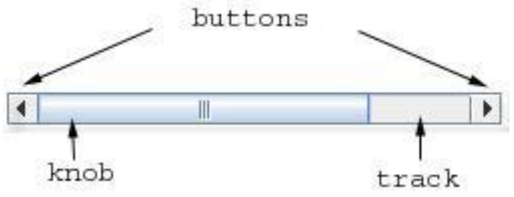
viewport, you can use `setViewportBorder`. A common operation to want to do is to set the background color that will be used if the main viewport view is smaller than the viewport, This can be accomplished by setting the

background color of the viewport, via .

Constructor	Purpose

JScrollPane()
JScrollPane(Component)
JScrollPane(int, int)
JScrollPane(Component, int, int)
	Create 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).





Method:

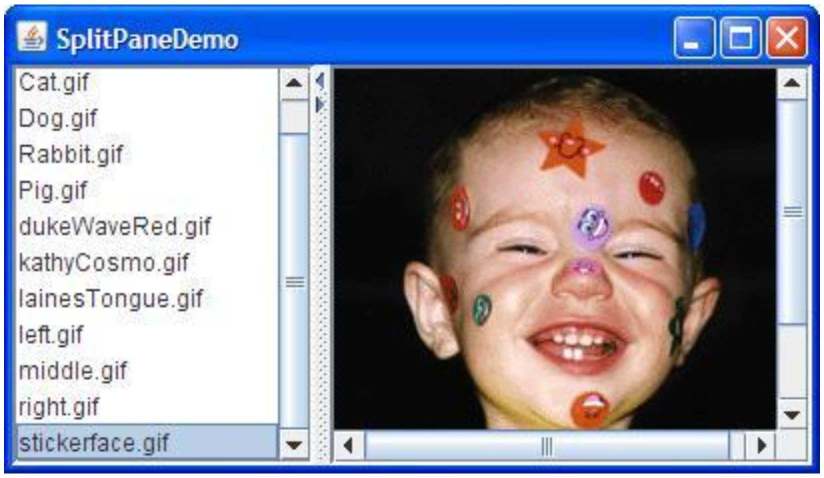
void setVerticalScrollBarPolicy(int) int getVerticalScrollBarPolicy()	Set or get the vertical scroll policy.ScrollPaneConstants defines three values for specifying this policy: VERTICAL_SCROLLBAR_AS_NEEDED(the default), VERTICAL_SCROLLBAR_ALWAYS, and VERTICAL_SCROLLBAR_NEVER.
void setHorizontalScrollBarPolicy(int) int getHorizontalScrollBarPolicy()	Set or get the horizontal scroll policy. ScrollPaneConstants defines three values for specifying this policy: HORIZONTAL_SCROLLBAR_AS_NEEDED(the default), HORIZONTAL_SCROLLBAR_ALWAYS, and HORIZONTAL_SCROLLBAR_NEVER.
void setColumnHeaderView(Component) void setRowHeaderView(Component)	Set the column or row header for the scroll pane.
void setCorner(String, Component) Component getCorner(String)	Set or get the corner specified. The intparameter specifies which corner and must be one of the following constants defined in ScrollPaneConstants: UPPER_LEFT_CORNER, UPPER_RIGHT_CORNER, and LOWER_LEFT_CORNER, LOWER_RIGHT_CORNER

3) Split Panes

A `JSplitPane` displays two components, either side by side or one on top of the other. By dragging the divider that appears between the components, the user can specify how much of the split pane's total area goes to each component. You can divide screen space among three or more components by putting split panes inside of split panes, as described in Nesting Split Panes.

Instead of adding the components of interest directly to a split pane, you often put each component into a scroll pane. You then put the scroll panes into the split pane. This allows the user to view any part of a component of interest, without requiring the component to take up a lot of screen space or adapt to displaying itself in varying amounts of screen space.

Here's a picture of an application that uses a split pane to display a list and an image side by side:



Constructor	Purpose

JSplitPane()
JSplitPane(int)
JSplitPane(int, boolean)
JSplitPane(int, Component, Component)
JSplitPane(int, boolean, Compone Component)
	Create a split pane. When present, the `int`parameter indicates the split pane's orientation, either`HORIZONTAL_SPLIT`(the default) or`VERTICAL_SPLI` The `boolean` parameter, when present, sets whether the components continually repaint as the user drags the split pane. If left unspecified, this option (called*continuous layout*) is turned off. The`Component`parameters set the initial left and right, top and bottom components, respectively.

Method :

void setOrientation(int) int getOrientation()	Set or get the split pane's orientation. Use either `HORIZONTAL_SPLIT` or `VERTICAL_SPLIT` defined in `JSplitPane`. If left unspecified, the split pane will be horizontall split.
void setDividerSize(int) int getDividerSize()	Set or get the size of the divider in pixels.
void setContinuousLayout(boolean) boolean isContinuousLayout()	Set or get whether the split pane's components ar continually layed out and painted while the user is dragging the divider. By default, continuous layout turned off.
void setOneTouchExpandable(boolean) boolean isOneTouchExpandable()	Set or get whether the split pane displays a control on the divider to expand/collapse the divider. The default depends on the look and feel. In the Java look and feel, it is off by default.
void add(Component)	Add the component to the split pane. You can add only two components to a split pane. The first component added is the top/left component. The

second component added is the bottom/right component. Any attempt to add more component results in an exception.

4)Tabbed Panes

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

Tabbed panes are encapsulated by the **JTabbedPane** class, which extends **JComponent**. We will use its default constructor.

The general procedure to use a tabbed pane in an applet is outlined here:
1. Create a **JTabbedPane** object.
2. Call **addTab( )** to add a tab to the pane. (The arguments to this method define the title of the tab and the component it contains.)
3. Repeat step 2 for each tab.
4. Add the tabbed pane to the content pane of the applet.

Constructor

```
JTabbedPane tp = new JTabbedPane();   // Tabs along the top edge.
JTabbedPane tp = new JTabbedPane(edge);
```

Where `edge` specifies which edge the tabs are on

- `JTabbedPane.TOP` (default)
- `JTabbedPane.RIGHT`
- `JTabbedPane.BOTTOM`
- `JTabbedPane.LEFT`

Constructor	Purpose
JTabbedPane()	Creates a tabbed pane. The first optional argument specifies where the tabs should appear. By default, the tabs appear at the

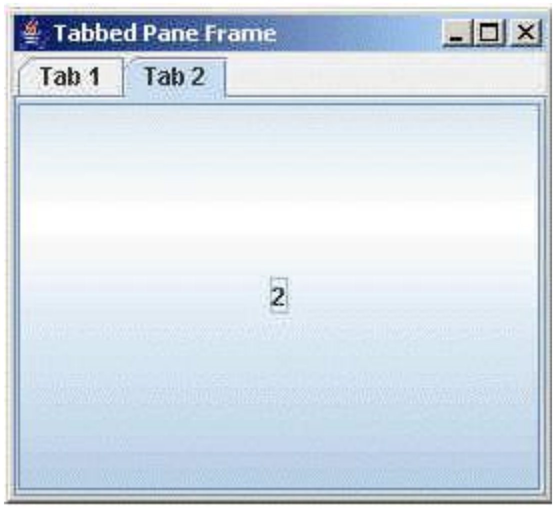
JTabbedPane(int)
JTabbedPane(int, int)
	top of the tabbed pane. You can specify these positions (defined in the `SwingConstants`interface,
	which `JTabbedPane`implements): `TOP`, `BOTTOM`, `LEFT`,`RIGHT`. The

	second optional argument specifies the tab layout policy. You can specify one of these policies (defined in`JTabbedPane`):`WRAP_TAB_LAYOUT` or`SCROLL_TAB_LAYOUT`.

Method

`Component`	`add(Component component)` Adds a `component` with a tab title defaulting to the name of the component which is the result of calling `component.getName`.
`Component`	`add(Component component, int index)` Adds a `component` at the specified tab index with a tab title defaulting to the name of the component.
`addTab(String, Icon, Component, String)` `addTab(String, Icon, Component)` `addTab(String, Component)`	**Adds a new tab to the tabbed pane. The first argument specifies the text on the tab. The optional icon argument specifies the tab's icon. The component argument specifies the component that the tabbed pane should show when the tab is selected. The fourth argument, if present, specifies the tool tip text for the tab.**
`insertTab(String, Icon, Component, String, int)`	**Inserts a tab at the specified index, where the first tab is at index 0. The arguments are the same as for addTab.**
`void setSelectedIndex(int)` `void setSelectedComponent(Component)`	**Selects the tab that has the specified component or index. Selecting a tab has the effect of displaying its associated component.**
`void setEnabledAt(int, boolean)` `boolean isEnabledAt(int)`	**Sets or gets the enabled state of the tab at the specified index.**

Example :



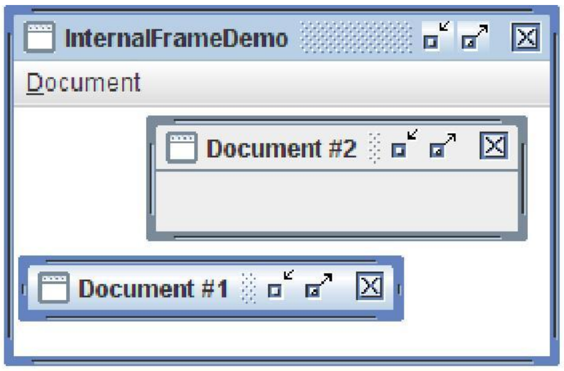
5) Internal Frame

A lightweight object that provides many of the features of a native frame, including dragging, closing, becoming an icon, resizing, title display, and support for a menu bar.

If you do not add the internal frame to a container (usually a `JDesktopPane`), the internal frame will not appear.

Generally, you add `JInternalFrame`s to a `JDesktopPane`. The UI delegates the look-and-feel-specific actions to the `DesktopManager` object maintained by the `JDesktopPane`.

Here is a picture of an application that has two internal frames (one of which is iconified) inside a regular frame:



Constructor Summary

`JInternalFrame()` Creates a non-resizable, non-closable, non-maximizable, non-iconifiable `JInternalFrame`with no title.
`JInternalFrame(String title)` Creates a non-resizable, non-closable, non-maximizable, non-iconifiable `JInternalFrame`with the specified title.
`JInternalFrame(String title, boolean resizable)` Creates a non-closable, non-maximizable, non-iconifiable `JInternalFrame` with the specified title and resizability.
`JInternalFrame(String title, boolean resizable, boolean closable)` Creates a non-maximizable, non-iconifiable `JInternalFrame` with the specified title, resizability, and closability.
`JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable)` Creates a non-iconifiable `JInternalFrame` with the specified title, resizability, closability, and maximizability.
`JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable, boolean iconifiable)` Creates a `JInternalFrame` with the specified title, resizability, closability, maximizability, and iconifiability.

Method	Purpose
void setVisible(boolean)	Make the internal frame visible (if`true`) or invisible (if `false`). You should invoke `setVisible(true)` on each `JInternalFrame` before adding it to its container. (Inherited from `Component`).
void pack()	Size the internal frame so that its components are at their preferred sizes.
void setLocation(Point) void setLocation(int, int)	Set the position of the internal frame. (Inherited from`Component`).
void setBounds(Rectangle) void setBounds(int, int, int, int)	Explicitly set the size and location of the internal frame. (Inherited from `Component`).
void setSize(Dimension) void setSize(int, int)	Explicitly set the size of the internal frame. (Inherited from`Component`).

3

USING ATOMC COMPONENTS

1)JLabel

A display area for a short text string or an image, or both. A label does not react to input events. As a result, it cannot get the keyboard focus. A label can, however, display a keyboard alternative as a convenience for a nearby component that has a keyboard alternative but can't display it.

A `JLabel` object can display either text, an image, or both. You can specify where in the label's display area the label's contents are aligned by setting the vertical and horizontal alignment. By default, labels are vertically centered in their display area. Text-only labels are leading edge aligned, by default; image-only labels are horizontally centered, by default.

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

`JLabel(String text, int horizontalAlignment)`

Creates a `JLabel` instance with the specified text and horizontal alignment.

Method Summary

void setIcon(Icon) Icon getIcon()	Sets or gets the image displayed by the label.
void setDisplayedMnemonic(char) char getDisplayedMnemonic()	Sets or gets the letter that should look like a keyboard alternative. This is helpful when a label describes a component (such as a text field) that has a keyboard alternative but cannot display it. If the labelFor property is also set (using setLabelFor), then when the user activates the mnemonic, the keyboard focus is transferred to the component specified by the labelFor property.
void setDisplayedMnemonicIndex(int) int getDisplayedMnemonicIndex()	Sets or gets a hint as to which character in the text should be decorated to represent the mnemonic. This is useful when you have two instances of the same character and wish to decorate the second instance. For example,setDisplayedMnemonicIndex(5)decorates the character that is at position 5 (that is, the 6th character in the text). Not all types of look and feel may support this feature.
void setDisabledIcon(Icon) Icon getDisabledIcon()	Sets or gets the image displayed by the label when it is disabled. If you do not specify a disabled image, then the look and feel creates one by manipulating the default image

2) **Class JButton**

Mr.Arvind Yadav +91 – 829 1020 333
arvind@objectzoom.com

JButton class provides the functionality of push button. JButton allows an icon , a string , or both to be associated with the push button .

Constructor Summary
`JButton()` Creates a button with no set text or icon.
`JButton(Action a)` Creates a button where properties are taken from the `Action` supplied.
`JButton(Icon icon)` Creates a button with an icon.
`JButton(String text)` Creates a button with text.
`JButton(String text, Icon icon)` Creates a button with initial text and an icon.

Method

void setAction(Action) Action getAction()	Set or get the button's properties according to values from the `Action` instance.
void setText(String) String getText()	Set or get the text displayed by the button. You can use HTML formatting, as described in Using HTML in Swing Components.
void setIcon(Icon) Icon getIcon()	Set or get the image displayed by the button when the button isn't selected or pressed.
void setDisabledIcon(Icon) Icon getDisabledIcon()	Set or get the image displayed by the button when it is disabled. If you do not specify a disabled image, then the look and feel creates one by manipulating the default image.
void setPressedIcon(Icon) Icon getPressedIcon()	Set or get the image displayed by the button when it is being pressed.

3)JCheckBox

An implementation of a check box -- an item that can be selected or deselected, and which displays its state to the user. By convention, any number of check boxes in a group can be selected.

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

Method:

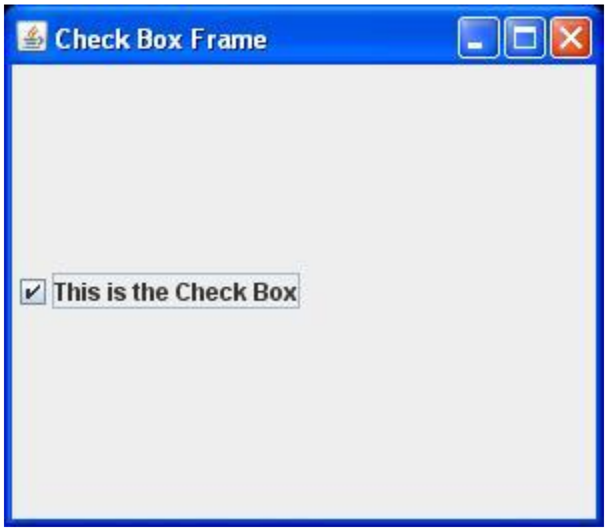
Void setSelected (Boolean state)

Set the state of the button here , state is true if the checkbox should be checked.

Check Boxes are created in swing by creating the instance of the **JCheckBox** class using it's constructor which contains the string which has to be shown beside the check box on the frame .

EXAMPLE

```
import javax.swing.*;

public class CreateCheckBox{
  public static void main(String[] args){
    JFrame frame = new JFrame("Check Box Frame");
    JCheckBox chk = new JCheckBox("This is the Check
    Box"); frame.add(chk);
    frame.setSize(400, 400);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```



4)JRadioButton

Constructor Summary
`JRadioButton()`   Creates an initially unselected radio button with no set text.

`JRadioButton``(Action a)`

Creates a radiobutton where properties are taken from the Action supplied.

`JRadioButton``(Icon icon)`

Creates an initially unselected radio button with the specified image but no text.
`JRadioButton(Icon icon, boolean selected)` Creates a radio button with the specified image and selection state, but no text.
`JRadioButton(String text)` Creates an unselected radio button with the specified text.
`JRadioButton(String text, boolean selected)` Creates a radio button with the specified text and selection state.
`JRadioButton(String text, Icon icon)` Creates a radio button that has the specified text and image, and that is initially unselected.
`JRadioButton(String text, Icon icon, boolean selected)` Creates a radio button that has the specified text, image, and selection state.

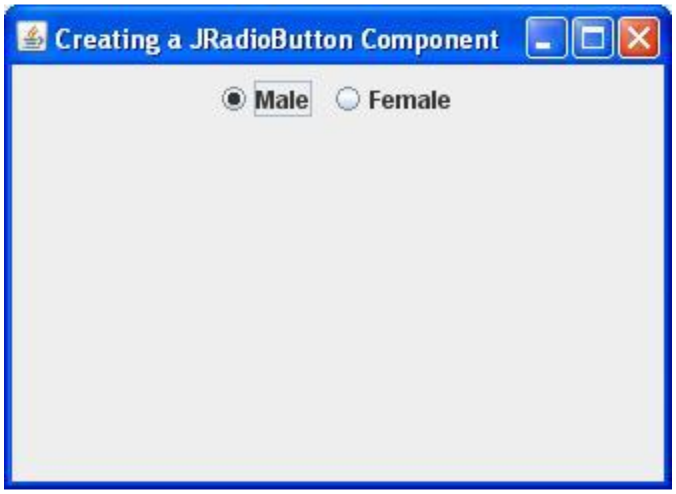
→ Radio buttons must be configured into a group.only one of the buttons in that group can be selected at any time.

→ The button group class is instantaiated to create a button group. Element are then added to the button group via the following. method void add(AbstractButton ab)
here ,ab is a referenceto the button to be added to the group.

how to create a radio button in java swing. Radio Button is like check box. Differences between check box and radio button are as follows:

1. Check Boxes are separated from one to another where Radio Buttons are the different-different button like check box from a same ButtonGroup.
2. You can checks multiple check boxes at once but this can never done in the case of radio button. You can select only one radio button at once from a group of the radio button.
3. You can check or uncheck the check box but you can on check the radio button by clicking it once.

Here, you will see the JRadioButton component creation procedure in java with the help of this program. This example provides two radio buttons same ButtonGroup. These radio buttons represent the option for choosing male or female. Following is the image for the result of the given program:



The creation of JRadioButton are completed by the following methods:

ButtonGroup:

This is the class of the *javax.swing.*;* package, which is used to create a group of radio buttons from which you can select only one option from that group of the radio buttons. This is class is used by creating a instance of if using it's constructor. Radio Buttons are added to the specified group using the add(JRadioButton) method of the **ButtonGroup** class.

JRadioButton:

This is the class has been used to create a single radio button for the application.

setSelected():

This method sets the value of the radio button. This method takes a boolean value either *true* or *false*. If you pass *true* value then the radio button will be selected otherwise the radio button is not selected.

Here is the code of program:

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

public class CreateRadioButton
{


  public CreateRadioButton()
  {
    JRadioButton Male,Female;
    JFrame frame = new JFrame("Creating a JRadioButton
    Component"); JPanel panel = new JPanel();
    ButtonGroup buttonGroup = new ButtonGroup();
    Male = new JRadioButton("Male");
```

```
buttonGroup.add(Male);
panel.add(Male);
Female = new JRadioButton("Female");
buttonGroup.add(Female);
panel.add(Female);
```

```
      Male.setSelected(true);
      frame.add(panel);
      frame.setSize(400,400);
      frame.setVisible(true);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    public static void main(String[] args)
    {
      CreateRadioButton r = new CreateRadioButton();
    }

  }
```

5)JCombo Boxes

Swing provides a *combo box* (a combination of a text field and a drop-down list) through the **JComboBox** class, which extends **JComponent**. A combo box normally displays one entry. However, it can also display a drop-down list that allows a user to select a different entry.

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

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

void addItem(Object *obj*)
Here, *obj* is the object to be added to the combo box

Method and Constructor	Purpose

JComboBox()
JComboBox(ComboBoxModel)
JComboBox(Object[])
JComboBox(Vector)
	Create a combo box with the specified items in its menu. A combo box created with the default constructor has no items in the menu initially. Each of the other constructors initializes the menu from its argument: a model object, an array of objects, or a `Vector` of objects.

void addItem(Object) void insertItemAt(Object, int)	Add or insert the specified object into the combo box's menu. The insert method places the specified object *at* the specified index, thus inserting it before the object currently at that index. These methods require that the combo box's data model be an instance of`MutableComboBoxModel`.
Object getItemAt(int) Object getSelectedItem()	Get an item from the combo box's menu.
void removeAllItems() void removeItemAt(int) void removeItem(Object)	Remove one or more items from the combo box's menu. These methods require that the combo box's data model be an instance of`MutableComboBoxModel`.
int getItemCount()	Get the number of items in the combo box's menu
void addActionListener(ActionListener)	Add an action listener to the combo box. The listener'sactionPerformedmethod is called when the user selects an item from the combo box's menu or, in an editable combo box, when the user presses Enter.
void addItemListener(ItemListener)	Add an item listener to the combo box. The listener'sitemStateChangedmethod is called when the selection state of any of the combo box's items change.

6)JList

A component that allows the user to select one or more objects from a list. A separate model,`ListModel`, represents the contents of the list.

Constructor Summary
`JList()` Constructs a `JList` with an empty model.
`JList(ListModel dataModel)` Constructs a `JList` that displays the elements in the specified, non-`null` model.
`JList(Object[] listData)` Constructs a `JList` that displays the elements in the specified array.
`JList(Vector listData)` Constructs a `JList` that displays the elements in the specified `Vector`.

JList

`public JList(ListModel dataModel)`

Constructs a `JList` that displays the elements in the specified, non-`null` model. All `JList`constructors delegate to this one.

Parameters:

`dataModel` - the data model for this list

Throws:

`IllegalArgumentException` - if `dataModel` is `null`

JList

`public JList(Object[] listData)`

Constructs a `JList` that displays the elements in the specified array. This constructor just delegates to the `ListModel` constructor.

Parameters:

`listData` - the array of Objects to be loaded into the data model

JList
`public JList(Vector listData)`

Constructs a `JList` that displays the elements in the specified `Vector`. This constructor just delegates to the `ListModel` constructor.

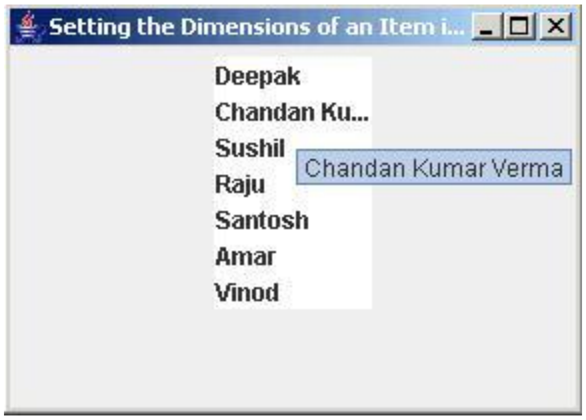
Parameters:

`listData` - the `Vector` to be loaded into the data model

JList
`public JList()`

Constructs a `JList` with an empty model.

Example:



7)JTextField

`JTextField` is a lightweight component that allows the editing of a single line of text.
The Swing text field is encapsulated by the **JTextComponent** class, which extends **JComponent**. It provides functionality that is common to Swing text components. One of its subclasses is **JTextField**, which allows you to edit one line of text.

A text field is a basic text control that enables the user to type a small amount of text. When the user indicates that text entry is complete (usually by pressing Enter), the text field fires an action event. If you need to obtain more than one line of input from the user, use a text area.

Constructor Summary
`JTextField()` Constructs a new `TextField`.
`JTextField(Document doc, String text, int columns)` Constructs a new `JTextField` that uses the given text storage model and the given number of columns.
`JTextField(int columns)` Constructs a new empty `TextField` with the specified number of columns.
`JTextField(String text)` Constructs a new `TextField` initialized with the specified text.
`JTextField(String text, int columns)` Constructs a new `TextField` initialized with the specified text and columns.

Method	Purpose
void setEditable(boolean) boolean isEditable() *(defined in* `JTextComponent`*)*	Sets or indicates whether the user can edit the text in the text field.
void setColumns(int); int getColumns()	Sets or obtains the number of columns displayed by the text field. This is really just a hint for computing the field's preferred width.
void setHorizontalAlignment(int); int getHorizontalAlignment()	Sets or obtains how the text is aligned horizontally within its area. You can use `JTextField.LEADING`, `JTextField.CENTER`, and `JTextField.TRAILING` for arguments.

8)JTextArea

A `JTextArea` is a multi-line area that displays plain text. It is intended to be a lightweight component that provides source compatibility with the `java.awt.TextArea` class where it can reasonably do so.

Constructor Summary
`JTextArea``()` Constructs a new TextArea.
`JTextArea``(Document doc)` Constructs a new JTextArea with the given document model, and defaults for all of the other arguments (null, 0, 0).
`JTextArea``(Document doc, String text, int rows, int columns)` Constructs a new JTextArea with the specified number of rows and columns, and the given model.

`JTextArea``(int rows, int columns)`

Constructs a new empty TextArea with the specified number of rows and

columns.
`JTextArea(String text)` Constructs a new TextArea with the specified text displayed.
`JTextArea(String text, int rows, int columns)` Constructs a new TextArea with the specified text and number of rows and columns.

```
public JTextArea()
```

Constructs a new TextArea. A default model is set, the initial string is null, and rows/columns are set to 0.

JTextArea

```
public JTextArea(String text)
```

Constructs a new TextArea with the specified text displayed. A default model is created and rows/columns are set to 0.

Parameters:

`text` - the text to be displayed, or null

JTextArea

```
public JTextArea(int rows, int
                 columns)
```

Constructs a new empty TextArea with the specified number of rows and columns. A default model is created, and the initial string is null.

Parameters:

`rows` - the number of rows >= 0

`columns` - the number of columns >= 0

Throws:

`IllegalArgumentException` - if the rows or columns arguments are negative.

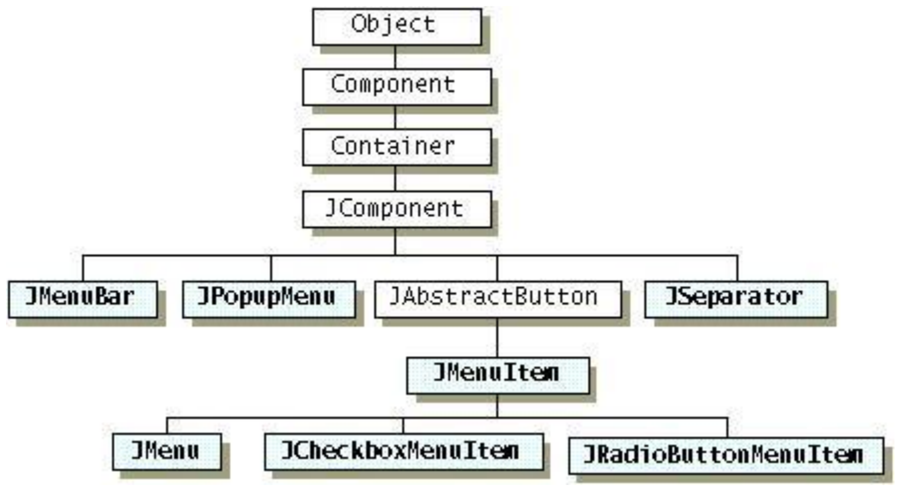
9)Menus

A menu provides a space-saving way to let the user choose one of several options.

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.

The Menu Component Hierarchy

Here is a picture of the inheritance hierarchy for the menu-related classes:



Classes involved:

JMenuBar()	Creates a menu bar.

JMenu()
JMenu(String)
JMenu(Action)
	Creates a menu. The string specifies the text to display for the menu. TheAction specifies the text and other properties of the menu

`JMenuItem()`	Creates a `JMenuItem` with no set text or icon.
`JCheckBoxMenuItem()`	Creates an initially unselected check box menu item with no set text or icon.
`JRadioButtonMenuItem()`	Creates a `JRadioButtonMenuItem` with no set text or icon

JMenuBar:

An implementation of a menu bar. You add `JMenu` objects to the menu bar to construct a menu. When the user selects a `JMenu` object, its associated `JPopupMenu` is displayed, allowing the user to select one of the `JMenuItems` on it.

.

JMenu:

An implementation of a menu -- a popup window containing `JMenuItem`s that is displayed when the user selects an item on the `JMenuBar`. In addition to `JMenuItem`s, a `JMenu` can also contain`JSeparator`s.

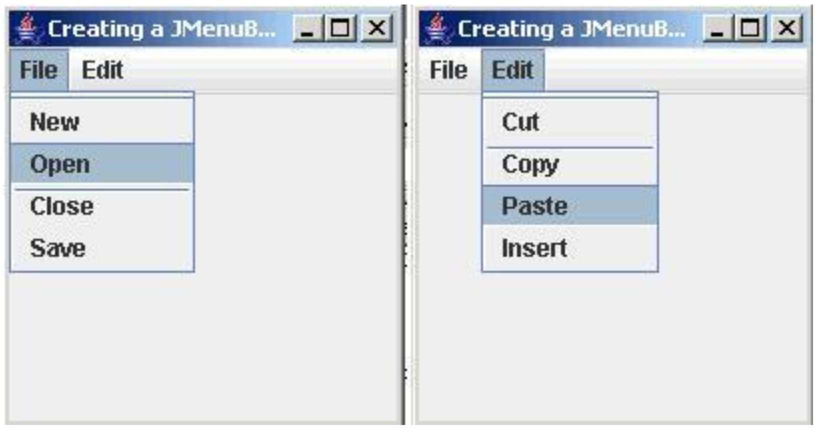
JCheckBoxMenuItem :

A menu item that can be selected or deselected. If selected, the menu item typically appears with a checkmark next to it. If unselected or deselected, the menu item appears without a checkmark. Like a regular menu item, a check box menu item can have either text or a graphic icon associated with it, or both.

JRadioButtonMenuItem:

An implementation of a radio button menu item. A `JRadioButtonMenuItem` is a menu item that is part of a group of menu items in which only one item in the group can be selected. The selected item displays its selected state. Selecting it causes any other selected item to switch to the unselected state. To control the selected state of a group of radio button menu items, use a `ButtonGroup` object.

Menu bar contains a collection of menus. Each menu can have multiple menu items these are called submenu. Similarly, all menus have multiples menu items. The Separator divides the menu items in a separate groups like same types of menu Items are divided into a individual parts. For pictorial representation, the image for the result of the given program is given below:



This program shows how to create menu bar, menus, submenus and Separators. Here, all items shows on a frame with the help of following methods and APIs:

JMenuBar:
This is the class which constructs a menu bar that contains several menus.

JMenu(String):
This is the constructor of **JMenu** class. This constructor constructs the new menu. It takes the string type value which is the name label for the menu.

JMenuItem(String):
This is the constructor of **JMenuItem** class which constructs new menu items for the specific menu. It takes string types value which is the label for the menu item.

JSeparator():
This is the constructor of **JSeparator** class which adds an extra line between menu items. This line, only separates the menu items.

setJMenuBar():
This method is used to set the menu bar to the specified frame. It takes the object of the **JMenuBar** class**.**

EXAMPLE:

```
import javax.swing.*;

public class SwingMenu

{

  public SwingMenu()

    {
     JFrame frame = new JFrame("MenuBar Menu MenuItm & seprator");
     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); JMenuBar
     menubar = new JMenuBar();
     JMenu filemenu = new JMenu("File");
     filemenu.add(new JSeparator());
     JMenu editmenu = new JMenu("Edit");
     editmenu.add(new JSeparator());
     JMenuItem fileItem1 = new JMenuItem("New");
     JMenuItem fileItem2 = new JMenuItem("Open");
     JMenuItem fileItem3 = new JMenuItem("Close");
     fileItem3.add(new JSeparator());
     JMenuItem fileItem4 = new JMenuItem("Save");
     JMenuItem editItem1 = new JMenuItem("Cut");
     JMenuItem editItem2 = new JMenuItem("Copy");
     editItem2.add(new JSeparator());
     JMenuItem editItem3 = new JMenuItem("Paste");
     JMenuItem editItem4 = new
     JMenuItem("Insert"); filemenu.add(fileItem1);
     filemenu.add(fileItem2);
     filemenu.add(fileItem3);
     filemenu.add(fileItem4);
     editmenu.add(editItem1);
     editmenu.add(editItem2);
     editmenu.add(editItem3);
     editmenu.add(editItem4);
     menubar.add(filemenu);
     menubar.add(editmenu);
     frame.setJMenuBar(menubar);
     frame.setSize(400,400);
     frame.setVisible(true);
     Public static void main (String [] args)
     {
        SwingMenu s = new SwingMenu();
     }
  }
}
```

10)JTables

The `JTable` is used to display and edit regular two-dimensional tables of cells

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

JTable(Object *data*[ ][ ], Object *colHeads*[ ])

Here, *data* is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings.

Here are the steps for using a table in an applet:

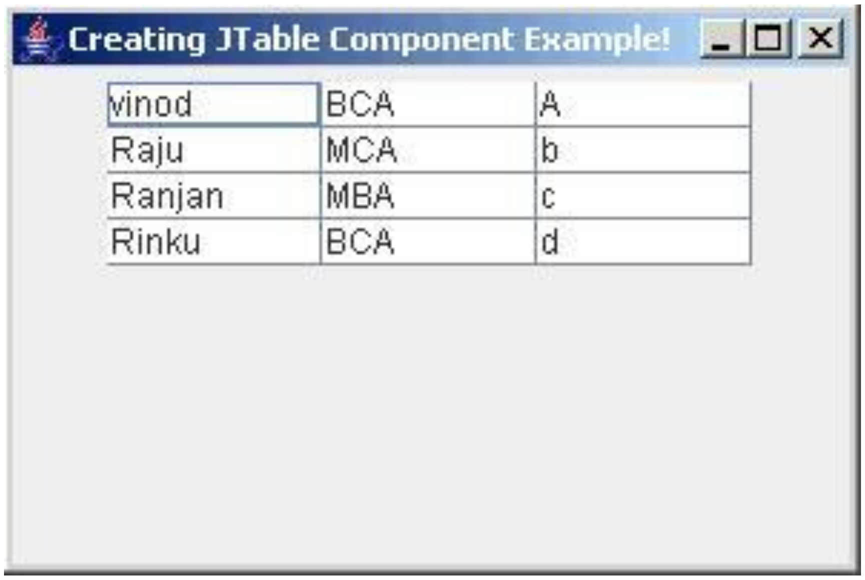
1. Create a **JTable** object.
2. Create a **JScrollPane** object. (The arguments to the constructor specify the table and the policies for vertical and horizontal scroll bars.)
3. Add the table to the scroll pane.
4. Add the scroll pane to the content pane of the applet

EXAMPLE:

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

public class JTableComponent
{

  public JTableComponent()
  {
    JFrame frame = new JFrame("Creating
    JTable Component Example!"); JPanel
    panel = new JPanel();
    String data[][] = {{"vinod","BCA","A"},{"Raju","MCA","b"},
     {"Ranjan","MBA","c"},{"Rinku","BCA","d"}};

    String col[] = {"Name","Course","Grade"};
    JTable table = new JTable(data,col);
    panel.add(table,BorderLayout.CENTER);
```

```
frame.add(panel);
frame.setSize(300,200);
frame.setVisible(true);
```

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

  }
}
```



vinod	BCA	A
Raju	MCA	b
Ranjan	MBA	c
Rinku	BCA	d

11) Trees

A *tree* is a component that presents a hierarchical view of data. A user has the ability to expand or collapse individual subtrees in this display. Trees are implemented in Swing by the **JTree** class, which extends **JComponent**. Some of its constructors are shown here:

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

The first form creates a tree in which each element of the hash table *ht* is a child node. Each element of the array *obj* is a child node in the second form. The tree node *tn* is the root of the tree in the third form. Finally, the last form uses the elements of vector *v* as child nodes.

A **JTree** object generates events when a node is expanded or collapsed. The **addTreeExpansionListener( )** and **removeTreeExpansionListener( )** methods allow listeners to register and unregister for these notifications. The signatures of these methods are shown here:

void addTreeExpansionListener(TreeExpansionListener *tel*) void removeTreeExpansionListener(TreeExpansionListener *tel)*

Here, *tel* is the listener object.

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

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

Here, *x* and *y* are the coordinates at which the mouse is clicked. The return value is a **TreePath** object that encapsulates information about the tree node that was selected by the user.

The **TreePath** class encapsulates information about a path to a particular node in a tree. It provides several constructors and methods. In this book, only the **toString( )** method is used. It returns a string equivalent of the tree path.

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

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

DefaultMutableTreeNode(Object *obj*)

Here, *obj* is the object to be enclosed in this tree node. The new tree node doesn't have a parent or children.

To create a hierarchy of tree nodes, the **add( )** method of **DefaultMutableTreeNode** can be used. Its signature is shown here:

void add(MutableTreeNode *child*)

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

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

TreePath getPath( )

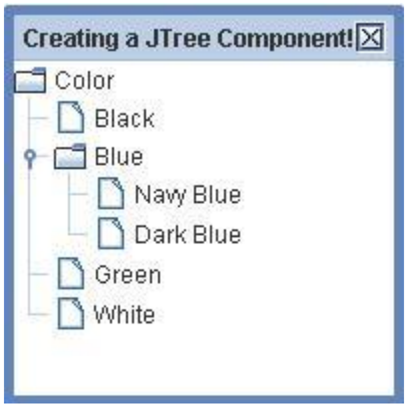
The **TreeExpansionListener** interface provides the following two methods:

void treeCollapsed(TreeExpansionEvent *tee*)
void treeExpanded(TreeExpansionEvent *tee*)

Here, *tee* is the tree expansion event. The first method is called when a subtree is hidden, and the second method is called when a subtree becomes visible.

EXAMPLE:

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

public class TreeComponent
{
  public static void main(String[] args)
  {
    JFrame frame = new JFrame("Creating a JTree Component!");
    DefaultMutableTreeNode parent = new DefaultMutableTreeNode("Color", true);
    DefaultMutableTreeNode black = new DefaultMutableTreeNode("Black");
    DefaultMutableTreeNode blue = new DefaultMutableTreeNode("Blue");
    DefaultMutableTreeNode nBlue = new DefaultMutableTreeNode("Navy Blue");
    DefaultMutableTreeNode dBlue = new DefaultMutableTreeNode("Dark Blue");
    DefaultMutableTreeNode green = new DefaultMutableTreeNode("Green");
    DefaultMutableTreeNode white = new DefaultMutableTreeNode("White");
    parent.add(black);
    parent.add(blue);
    blue.add(nBlue);
    blue.add(dBlue);
    parent.add(green );
    parent.add(white);
    JTree tree = new JTree(parent);
    frame.add(tree);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setUndecorated(true);
    frame.getRootPane().setWindowDecorationStyle(JRootPane.PLAIN_DIALOG);
    frame.setSize(200,200);
    frame.setVisible(true);
  }
}
```



12)JProgressBar

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

To indicate that a task of unknown length is executing, you can put a progress bar into indeterminate mode. While the bar is in indeterminate mode, it animates constantly to show that work is occurring. As soon as you can determine the task's length and amount of progress, you should update the progress bar's value and switch it back to determinate mode.

`JProgressBar` uses a `BoundedRangeModel` as its data model, with the `value` property representing the "current" state of the task, and the `minimum` and `maximum` properties representing the beginning and end points, respectively.

Constructor	Purpose

JProgressBar()
JProgressBar(int, int)
	Create a horizontal progress bar. The no-argument constructor initializes the progress bar with a minimum and initial value of 0 and a maximum of 100. The constructor with two integer arguments specifies the minimum and maximum values.

JProgressBar(int)
JProgressBar(int, int, int)
	Create a progress bar with the specified orientation, which can be either`JProgressBar.HORIZONTAL`or`JProgressBar.VERTICAL`. The optional second and third arguments specify minimum and maximum values.

JProgressBar(BoundedRangeModel)	Create a horizontal progress bar with the specified range model.

void setOrientation(int) int getOrientation()

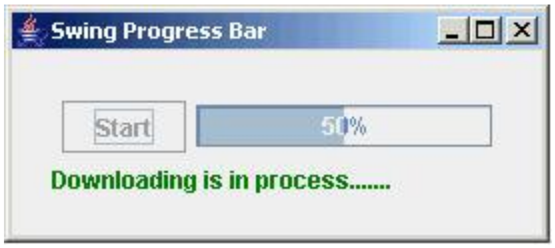
Method	
void setValue(int) int getValue()	
void setMinimum(int) int getMinimum()	
void setMaximum(int) int getMaximum()	

orJProgressBar.HORIZONTAL.

P u r p o s e

Set or get the current value of the progress bar. The value is constrained by the minimum and maximum values.
Set or get the minimum value of the progress bar.
Set or get the maximum value of the progress bar.
Set or get whether the progress bar is vertical or horizontal. Acceptable values are JProgressBar.VERTICAL

EXAMPLE:



JProgressBar:

This is the class which creates the progress bar using it's constructor **JProgressBar()** to show the status of your process completion. The constructor **JProgressBar()** takes two argument as parameter in which, first is the initial value of the progress bar which is shown in the starting and another argument is the counter value by which the value of the progress bar is incremented. Here, the value of the progress bar is incremented by 20.

setStringPainted(boolean):

This is the method of the **JProgressBar** class which shows the complete process in percent on the progress bar. It takes a boolean value as a parameter. If you pass the *true* then the value will be seen on the progress bar otherwise not seen.

setValue():

This is the method of the **JProgressBar** class which sets the value to the progress bar.

Timer():

This the constructor of the **Timer** class which starts the timer for timing. This constructor takes two argument as parameter first is the interval (in milliseconds) of the timer and second one is the listener object. Time is started using the start() method of the **Timer** class.

Here is the code of the program:

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


public class SwingProgressBar{
    int interval = 1000;
    int i; JLabel
    label;
    JProgressBar pb;
    Timer timer;
    JButton button;

    public SwingProgressBar()
    {
    JFrame frame = new JFrame("Swing Progress
    Bar"); button = new JButton("Start");
    button.addActionListener(new ButtonListener());

    pb = new JProgressBar(0, 20);
    pb.setValue(0);
    pb.setStringPainted(true);

    label = new JLabel("Arvind soft infotech");

    JPanel panel = new JPanel();
        panel.add(button);
        panel.add(pb);

        JPanel panel1 = new JPanel();
        panel1.setLayout(new BorderLayout());
        panel1.add(panel, BorderLayout.NORTH);
        panel1.add(label, BorderLayout.CENTER);
        panel1.setBorder(BorderFactory.createEmptyBorder(20, 20, 2
0, 20));
        frame.setContentPane(panel1);
        frame.pack();
        frame.setVisible(true);
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Create a timer.
        timer = new Timer(interval, new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
```

```
        if (i == 20){
          Toolkit.getDefaultToolkit().beep();
          timer.stop();
          button.setEnabled(true);
```

```
            pb.setValue(0);
            String str = "Downloading completed." ;
            label.setText(str);
          }
          i = i + 1;
                pb.setValue(i);
              }
          });
      }

      class ButtonListener implements ActionListener {
          public void actionPerformed(ActionEvent ae) {
              button.setEnabled(false);
        i = 0;
        String str = "Downloading is in
        process”; label.setText(str);
              timer.start();
          }
      }

      public static void main(String[] args) {
          SwingProgressBar spb = new SwingProgressBar();
      }
}
```

4

Layout Manager

A layout manager is an object that implements the `LayoutManager` interface and determines the size and position of the components within a container. Although components can provide size and alignment hints, a container's layout manager has the final say on the size and position of the components within the container.
a layout manager automatically arranges your controls within a window

Each **Container** object has a layout manager associated with it. A layout manager is an instance of any class that implements the **LayoutManager** interface. The layout manager is set by the **setLayout( )** method. If no call to **setLayout( )** is made, then the default layout manager is used. Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it.
The **setLayout( )** method has the following general form:

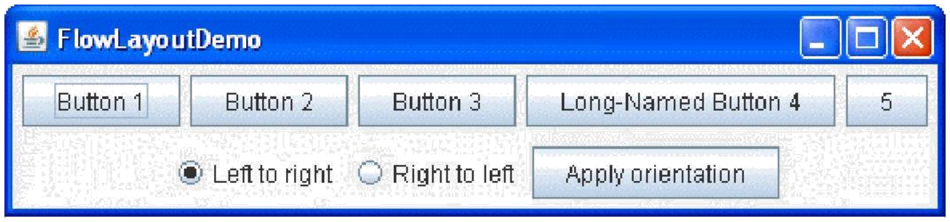
void setLayout(LayoutManager *layoutObj*)

Here, *layoutObj* is a reference to the desired layout manager. If you wish to disable the layout manager and position components manually, pass **null** for *layoutObj.* If you do this, you will need to determine the shape and position of each component manually, using the **setBounds( )** method defined by **Component**. Normally, you will want to use a layout manager.

1)FlowLayout

The `FlowLayout` class provides a very simple layout manager that is used, by default, by the `JPanel` objects

The `FlowLayout` class puts components in a row, sized at their preferred size. If the horizontal space in the container is too small to put all the components in one row, the`FlowLayout` class uses multiple rows. If the container is wider than necessary for a row of components, the row is, by default, centered horizontally within the container. To specify that the row is to aligned either to the left or right, use a `FlowLayout` constructor that takes an alignment argument. Another constructor of the `FlowLayout` class specifies how much vertical or horizontal padding is put around the components.



Constructor	Purpose
`FlowLayout()`	Constructs a new `FlowLayout` object with a centered alignment and horizontal and vertical gaps with the default size of 5 pixels.
`FlowLayout(intalign)`	Creates a new flow layout manager with the indicated alignment and horizontal and vertical gaps with the default size of 5 pixels. The alignment argument can be`FlowLayout.LEADING`,`FlowLayout.CENTER`, or`FlowLayout.TRAILING`. When the`FlowLayout` object controls a container with a left-to right component orientation (the default), the `LEADING`value specifies the components to be left-aligned and the `TRAILING` value specifies the components to be right-aligned.
`FlowLayout (int align, int hgap, intvgap)`	Creates a new flow layout manager with the indicated alignment and the indicated horizontal and vertical gaps. The `hgap` and `vgap` arguments specify the number of pixels to put between components.

2)BorderLayout

A border layout lays out a container, arranging and resizing its components to fit in five regions: north, south, east, west, and center. Each region may contain no more than one component, and is identified by a corresponding constant: `NORTH`, `SOUTH`, `EAST`, `WEST`, and `CENTER`. When adding a component to a container with a border layout, use one of these five constants, for example:

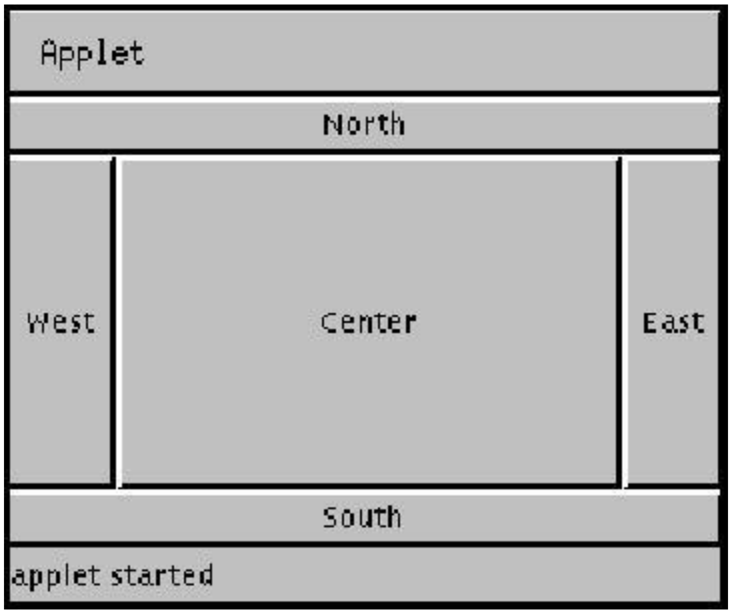
```
    Panel p = new Panel();
    p.setLayout(new BorderLayout());
    p.add(new Button("Okay"), BorderLayout.SOUTH);
```

The components are laid out according to their preferred sizes and the constraints of the container's size. The `NORTH` and `SOUTH` components may be stretched horizontally;

the `EAST` and `WEST`components may be stretched vertically; the `CENTER` component may stretch both horizontally and vertically to fill any space left over.

EXAMPLE:

```
import java.awt.*;
 import java.applet.Applet;

 public class buttonDir extends Applet
   { public void init() {
     setLayout(new BorderLayout());
     add(new Button("North"), BorderLayout.NORTH);
     add(new Button("South"), BorderLayout.SOUTH);
     add(new Button("East"), BorderLayout.EAST);
     add(new Button("West"), BorderLayout.WEST);
     add(new Button("Center"), BorderLayout.CENTER);
   }
 }
```



3)GridBagLayout

The `GridLayout` class is a layout manager that lays out a container's components in a rectangular grid. The container is divided into equal-sized rectangles, and one component is placed in each rectangle.

For example, the following is an applet that lays out six buttons into three rows and two columns:

```
import java.awt.*;
import java.applet.Applet;
public class ButtonGrid extends Applet
        { public void init()
   {
        setLayout(new GridLayout(3,2));
        add(new Button("1"));
        add(new  Button("2"));
        add(new  Button("3"));
        add(new  Button("4"));
        add(new  Button("5"));
        add(new Button("6"));
    }
}
```

If the container's `ComponentOrientation` property is horizontal and left-to-right, the above example produces the output shown in Figure 1. If the container's `ComponentOrientation` property is horizontal and right-to-left, the example produces the output shown in Figure 2.

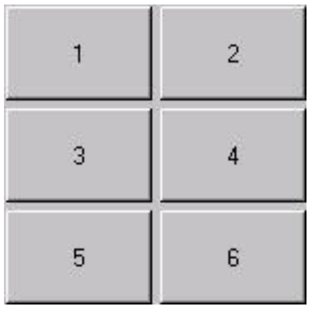


Figure 1: Horizontal, Left-to-Right

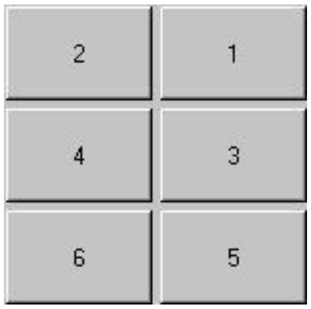


Figure 2: Horizontal, Right-to-Left

When both the number of rows and the number of columns have been set to non-zero values, either by a constructor or by the `setRows` and `setColumns` methods, the number of columns specified is ignored. Instead, the number of columns is determined from the specified number or rows and the total number of components in the layout. So, for example, if three rows and two columns have been specified and nine components are added to the layout, they will be displayed as three rows of three columns. Specifying the number of columns affects the layout only when the number of rows is set to zero.

Constructor	Purpose
`GridLayout(introws, intcols)`	**Creates a grid layout with the specified number of rows and columns. All components in the layout are given equal size. One, but not both, of `rows`and `cols` can be zero, which means that any number of objects can be placed in a row or in a column.**
`GridLayout(introws, intcols, inthgap, intvgap)`	**Creates a grid layout with the specified number of rows and columns. In addition, the horizontal and vertical gaps are set to the specified values. Horizontal gaps are places between each of columns. Vertical gaps are placed between each of the rows.**

4)GridBagLayout

The `GridBagLayout` class is a flexible layout manager that aligns components vertically and horizontally, without requiring that the components be of the same size.
Each `GridBagLayout` object maintains a dynamic, rectangular grid of cells, with each component occupying one or more cells, called its *display area*.

Each component managed by a `GridBagLayout` is associated with an instance of`GridBagConstraints`. The constraints object specifies where a component's display area should be located on the grid and how the component should be positioned within its display area. In addition to its constraints object, the `GridBagLayout` also considers each component's minimum and preferred sizes in order to determine a component's size.

To use a grid bag layout effectively, you must customize one or more of the `GridBagConstraints`objects that are associated with its components. You customize a `GridBagConstraints` object by setting one or more of its instance variables:

To use a grid bag layout effectively, you must customize one or more of the `GridBagConstraints`objects that are associated with its components. You customize a `GridBagConstraints` object by setting one or more of its instance variables:

`GridBagConstraints.gridx`, `GridBagConstraints.gridy`

> Specifies the cell containing the leading corner of the component's display area, where the cell at the origin of the grid has address `gridx = 0`, `gridy = 0`. For horizontal left-to-right layout, a component's leading corner is its upper left. For horizontal right-to-left layout, a component's leading corner is its upper right.
> Use `GridBagConstraints.RELATIVE` (the default value) to specify that the component be placed immediately following (along the x axis for `gridx` or the y axis for `gridy`) the component that was added to the container just before this component was added.

`GridBagConstraints.gridwidth`, `GridBagConstraints.gridheight`

> Specifies the number of cells in a row (for `gridwidth`) or column (for `gridheight`) in the component's display area. The default value is 1.
> Use `GridBagConstraints.REMAINDER` to specify that the component's display area will be from `gridx` to the last cell in the row (for`gridwidth`) or from `gridy` to the last cell in the column (for `gridheight`). Use`GridBagConstraints.RELATIVE` to

specify that the component's display area will be from`gridx` to the next to the last cell in its row (for `gridwidth` or from `gridy` to the next to the last cell in its column (for `gridheight`).

`GridBagConstraints.fill`

Used when the component's display area is larger than the component's requested size to determine whether (and how) to resize the component. Possible values are`GridBagConstraints.NONE` (the default), `GridBagConstraints.HORIZONTAL` (make the component wide enough to fill its display area horizontally, but don't change its height),`GridBagConstraints.VERTICAL` (make the component tall enough to fill its display area vertically, but don't change its width), and `GridBagConstraints.BOTH` (make the component fill its display area entirely).

`GridBagConstraints.ipadx`, `GridBagConstraints.ipady`

Specifies the component's internal padding within the layout, how much to add to the minimum size of the component. The width of the component will be at least its minimum width plus`ipadx` pixels. Similarly, the height of the component will be at least the minimum height plus`ipady` pixels.

`GridBagConstraints.insets`

Specifies the component's external padding, the minimum amount of space between the component and the edges of its display area.

`GridBagConstraints.anchor`

Used when the component is smaller than its display area to determine where (within the display area) to place the component. There are two kinds of possible values: relative and absolute. Relative values are interpreted relative to the container's `ComponentOrientation`property while absolute values are not. Valid values are:

Absolute Values

- **`GridBagConstraints.NORTH`**
- **`GridBagConstraints.SOUTH`**
- **`GridBagConstraints.WEST`**
- **`GridBagConstraints.EAST`**
- **`GridBagConstraints.NORTHWEST`**
- **`GridBagConstraints.NORTHEAST`**
- **`GridBagConstraints.SOUTHWEST`**
- **`GridBagConstraints.SOUTHEAST`**
- **`GridBagConstraints.CENTER` (the default)**

Relative Values

- **`GridBagConstraints.PAGE_START`**
- **`GridBagConstraints.PAGE_END`**
- **`GridBagConstraints.LINE_START`**
- **`GridBagConstraints.LINE_END`**
- **`GridBagConstraints.FIRST_LINE_START`**
- **`GridBagConstraints.FIRST_LINE_END`**
- **`GridBagConstraints.LAST_LINE_START`**
- **`GridBagConstraints.LAST_LINE_END`**

`GridBagConstraints.weightx`, `GridBagConstraints.weighty`

Used to determine how to distribute space, which is important for specifying resizing behavior. Unless you specify a weight for at least one component in a row (`weightx`) and column (`weighty`), all the components clump together in the center of their container. This is because when the weight is zero (the default), the `GridBagLayout` object puts any extra space between its grid of cells and the edges of the container.

The following figures show ten components (all buttons) managed by a grid bag layout. Figure 1 shows the layout for a horizontal, left-to-right container and Figure 2 shows the layout for a horizontal, right-to-left container.

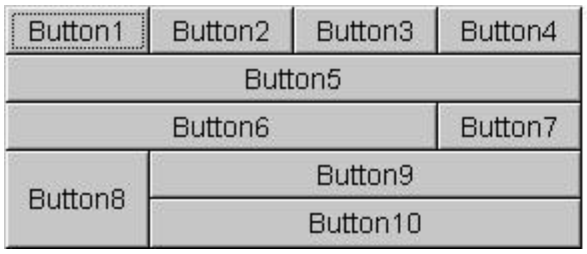


Figure 1: Horizontal, Left-to-Right

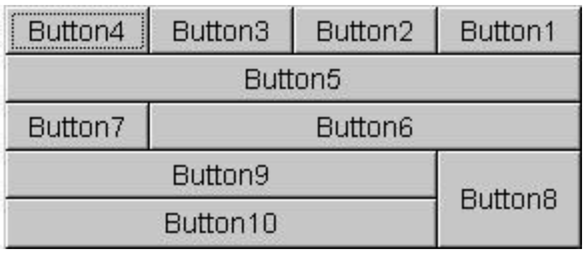


Figure 2: Horizontal, Right-to-Left

Each of the ten components has associated `GridBagConstraints` object set to`GridBagConstraints.BOTH`. In addition, the components have the following non-default constraints: the `fill` field of its `weightx = 1.0`

- Button1, Button2, Button3:
- Button4: `weightx = 1.0`, `gridwidth = GridBagConstraints.REMAINDER`
- Button5: `gridwidth = GridBagConstraints.REMAINDER`
- Button6: `gridwidth = GridBagConstraints.RELATIVE`
- Button7: `gridwidth = GridBagConstraints.REMAINDER`
- Button8: `gridheight = 2`, `weighty = 1.0`
- Button9, Button 10: `gridwidth = GridBagConstraints.REMAINDER`

Here is the code that implements the example shown above:

```
import java.awt.*;
 import java.util.*;
 import java.applet.Applet;

 public class GridBagEx1 extends Applet {


     protected void makebutton(String name,
                               GridBagLayout gridbag,
                                GridBagConstraints c) {
         Button button = new Button(name);
         gridbag.setConstraints(button, c);
         add(button);
     }
```

```
public void init() {
    GridBagLayout gridbag = new GridBagLayout();
    GridBagConstraints c = new GridBagConstraints();

    setFont(new Font("SansSerif", Font.PLAIN, 14));
    setLayout(gridbag);

    c.fill = GridBagConstraints.BOTH;
    c.weightx = 1.0;
    makebutton("Button1", gridbag, c);
    makebutton("Button2", gridbag, c);
    makebutton("Button3", gridbag, c);

      c.gridwidth = GridBagConstraints.REMAINDER; //end
    row makebutton("Button4", gridbag, c);

    c.weightx = 0.0;                   //reset to the default
     makebutton("Button5", gridbag, c); //another row

   c.gridwidth = GridBagConstraints.RELATIVE; //next-to-last in row
    makebutton("Button6", gridbag, c);

      c.gridwidth = GridBagConstraints.REMAINDER; //end
    row makebutton("Button7", gridbag, c);

      c.gridwidth = 1;                //reset to the default
      c.gridheight = 2;
    c.weighty = 1.0;
    makebutton("Button8", gridbag, c);

    c.weighty = 0.0;                  //reset to the default
      c.gridwidth = GridBagConstraints.REMAINDER; //end row
      c.gridheight = 1;               //reset to the default
    makebutton("Button9", gridbag, c);
    makebutton("Button10", gridbag, c);

    setSize(300, 100);
}

public static void main(String args[]) {
      Frame f = new Frame("GridBag Layout
      Example"); GridBagEx1 ex1 = new GridBagEx1();

      ex1.init();

      f.add("Center",
      ex1); f.pack();
```

```
            f.setSize(f.getPreferredSize());
            f.show();
      }
 }
```

5)CardLayout

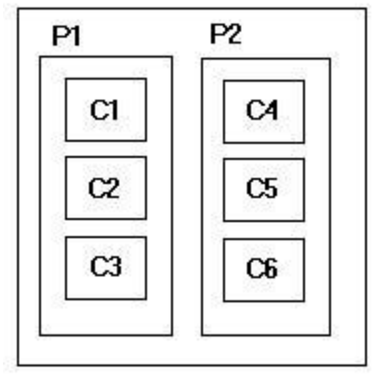
A `CardLayout` object is a layout manager for a container. It treats each component in the container as a card. Only one card is visible at a time, and the container acts as a stack of cards. The first component added to a `CardLayout` object is the visible component when the container is first displayed.

The ordering of cards is determined by the container's own internal ordering of its component objects.`CardLayout` defines a set of methods that allow an application to flip through these cards sequentially, or to show a specified card.
The `addLayoutComponent(java.awt.Component, java.lang.Object)` method can be used to associate a string identifier with a given card for fast random access.

Method	Purpose
`first (Containerparent)`	Flips to the first card of the container.
`next (Containerparent)`	Flips to the next card of the container. If the currently visible card is the last one, this method flips to the first card in the layout.
`previous (Containerparent)`	Flips to the previous card of the container. If the currently visible card is the first one, this method flips to the last card in the layout.
`last (Containerparent)`	Flips to the last card of the container.
`show (Containerparent, Stringname)`	Flips to the component that was added to this layout with the specified `name`, using the `addLayoutComponent` method.

6)BoxLayout

A layout manager that allows multiple components to be laid out either vertically or horizontally. The components will not wrap so, for example, a vertical arrangement of components will stay vertically arranged when the frame is resized.

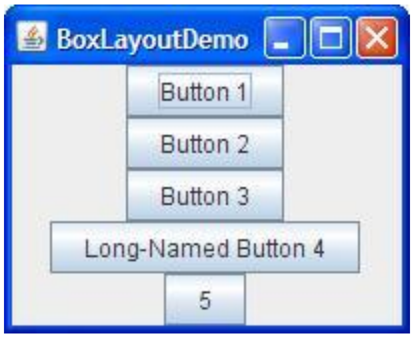


Nesting multiple panels with different combinations of horizontal and vertical gives an effect similar to GridBagLayout, without the complexity. The diagram shows two panels arranged horizontally, each of which contains 3 components arranged vertically.

The BoxLayout manager is constructed with an axis parameter that specifies the type of layout that will be done. There are four choices:

`X_AXIS` - Components are laid out horizontally from left to right.

`Y_AXIS` - Components are laid out vertically from top to bottom.



The `BoxLayout` class puts components in a single row or column. It respects the components' requested maximum sizes and also lets you align components

5

Event Handling

The user communicates with the programs by programming action like Clicking on a button and this action result in generation of events. The event are object that describe what has happened, the process of responding to an event is known as event handling

Event Source:
The object which generated the event is known as event source .

E.g.: mouse click on a button component generate an ActionEvent with the button as the source of the event .

Event Handler:

The method which receives the event, process the event and does something on the event being generated is known as the Event Handler.The Event Handler are the methods, which are within the Event Listeners in the Event Delegation Model followed by java.

The Event class defines the list of events by programs and also provide the constructors for constructing event but we does not need to use this constructor as the event are internally generated by the java run time system in response to user interface action .

Implementing Listeners for Commonly Handled Events:

1) Action Listener:

Action listeners are probably the easiest — and most common — event handlers to implement. You implement an action listener to define what should be done when an user performs certain operation.

An action event occurs, whenever an action is performed by the user. Examples: When the user clicks a button, chooses a menu item, presses Enter in a text field. The result is that an `actionPerformed` message is sent to all action listeners that are registered on the relevant component.

To write an Action Listener, follow the steps given below:

1. Declare an event handler class and specify that the class either implements an ActionListener interface or extends a class that implements an ActionListener interface. For example:

   ```
   public class MyClass implements ActionListener {
   ```
2. Register an instance of the event handler class as a listener on one or more components. For example:

   ```
   someComponent.addActionListener(instanceOfMyClass);
   ```

3. Include code that implements the methods in listener interface. For example:

   ```
   public void actionPerformed(ActionEvent e)
   {
       ...//code that reacts to the action...
   }
   ```

The ActionListener Interface

Method	Purpose
actionPerformed(actionEvent)	Called just after the user performs an action

The ActionEvent Class

Method	**Purpose**
String getActionCommand()	Returns the string associated with this action. Most objects that can fire action events support a method called `setActionCommand` that lets you set this string.
int getModifiers()	Returns an integer representing the modifier keys the user was pressing when the action event occurred. You can use the `ActionEvent`-defined constants `SHIFT_MASK`, `CTRL_MASK`, `META_MASK`, and `ALT_MASK` to determine which keys were pressed. For example, if the user Shift-selects a menu item, then the following expression is nonzero: `actionEvent.getModifiers() & ActionEvent.SHIFT_MASK`
Object getSource()	Returns the object that fired the event.

2)Component Listener

The Component listener is a listener interface for receiving component events. A component is an object having a graphical representation that can be displayed on the screen and that can interact with the user. Some of the examples of components are the buttons, checkboxes, and scrollbars of a typical graphical user interface.

The class that is interested in processing a component event either implements this interface and all the methods it contains, or extends the abstract ComponentAdapter class overriding only the methods of interest. The listener object created from that class is then registered with a component using the component's addComponentListener method. When the component's size, location, or visibility changes, the relevant method in the listener object is invoked, and the ComponentEvent is passed to it.

The ComponentListener Interface.

Method	Purpose
componentHidden(ComponentEvent)	Called after the listened-to component is hidden as the result of the`setVisible`method being called.
componentMoved(ComponentEvent)	Called after the listened-to component moves, relative to its container. For example, if a window is moved, the window fires a component-moved event, but the components it contains do not.
componentResized(ComponentEvent)	Called after the listened-to component's size (rectangular bounds) changes.
componentShown(ComponentEvent)	Called after the listened-to component becomes visible as the result of the`setVisible`method being called.

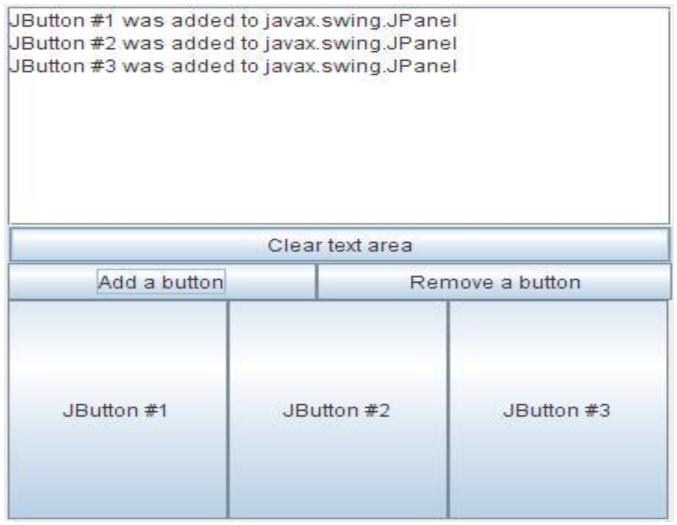
The ComponentEvent Class

Method	Purpose
Component getComponent()	Returns the component that fired the event. You can use this instead of the`getSource` method.

3) Container Listener

Container events are fired by a `Container` just after a component is added to or removed from the container. These events are for notification only — no container listener need be present for components to be successfully added or removed.

The following example demonstrates container events. By clicking **Add a button** or **Remove a button**, you can add buttons to or remove them from a panel at the bottom of the window. Each time a button is added to or removed from the panel, the panel fires a container event, and the panel's container listener is notified. The listener displays descriptive messages in the text area at the top of the window.



```
JButton #1 was added to javax.swing.JPanel
JButton #2 was added to javax.swing.JPanel
JButton #3 was added to javax.swing.JPanel
```

The ContainerListener Interface

The corresponding adapter class is `ContainerAdapter`

Method	Purpose
componentAdded(ContainerEvent)	Called just after a component is added to the listened-to container.
	Called just after a component is removed from

componentRemoved(ContainerEvent)	the listened-to container.

The ContainerEvent Class

Method	Purpose
Component getChild()	Returns the component whose addition or removal triggered this event.
Container getContainer()	Returns the container that fired this event. You can use this instead of the `getSource`method.

4) Focus Listener

Focus events are fired whenever a component gains or loses the keyboard focus. This is true whether the change in focus occurs through the mouse, the keyboard, or programmatically

The FocusListener Interface

The corresponding adapter class is `FocusAdapter`.

Method	Purpose
focusGained(FocusEvent)	Called just after the listened-to component gets the focus.
focusLost(FocusEvent)	Called just after the listened-to component loses the focus.

The FocusEvent API

Method	Purpose
boolean isTemporary()	Returns the true value if a focus-lost or focus-gained event is temporary.
Component getComponent()	Returns the component that fired the focus event.

5)Internal Frame Listener

An InternalFrameListener is similar to aWindowListener. Like the window listener, the internal frame listener listens for events that occur when the "window" has been shown for the first time, disposed of, iconified, deiconified, activated, or deactivated.

The InternalFrameListener Interface

The corresponding adapter class is**InternalFrameAdapter**.

Method	Purpose
internalFrameOpened (InternalFrameEvent)	(InternalFrameEvent) internalFrameDeiconified
internalFrameClosing (InternalFrameEvent)	
internalFrameClosed (InternalFrameEvent)	
internalFrameIconified	

Called just after the listened-to internal frame has been shown for the first time.	Called just after the listened-to internal frame has been disposed of.
Called in response to a user request that the listened-to internal frame be closed. By default, `JInternalFrame`hides the window when the user closes it. You can use the `JInternalFramesetDefaultCloseOperation`method to specify another option, which must be either`DISPOSE_ON_CLOSE` or`DO_NOTHING_ON_CLOSE` (both defined in `WindowConstants`, an interface that`JInternalFrame`implements). Or by implementing an`internalFrameClosing`method in the internal frame's listener, you can add custom behavior (such as bringing up dialogs or saving data) to internal frame closing.	Called just after the listened-to internal frame is iconified or deiconified, respectively.

(InternalFrameEvent)	
internalFrameActivated (InternalFrameEvent) internalFrameDeactivated(InternalFrameEvent)	Called just after the listened-to internal frame is activated or deactivated, respectively.

Each internal frame event method has a single parameter: an `InternalFrameEvent` object. The`InternalFrameEvent` class defines no generally useful methods. To get the internal frame that fired the event, use the `getSource` method, which `InternalFrameEvent`inherits from `java.util.EventObject`.

6)Mouse Listener

Mouse events notify when the user uses the mouse (or similar input device) to interact with a component. Mouse events occur when the cursor enters or exits a component's onscreen area and when the user presses or releases one of the mouse buttons.

Tracking the cursor's motion involves significantly more system overhead than tracking other mouse events. That is why mouse-motion events are separated into Mouse Motion listener type

The MouseListener Interface

Method	**Purpose**
mouseClicked(MouseEvent)	Called just after the user clicks the listened-to component.
mouseEntered(MouseEvent)	Called just after the cursor enters the bounds of the listened-to component.
mouseExited(MouseEvent)	Called just after the cursor exits the bounds of the listened-to component.
mousePressed(MouseEvent)	Called just after the user presses a mouse button while the cursor is over the listened-to component.

mouseReleased(MouseEvent)	Called just after the user releases a mouse button after a mouse press over the listened-to component.

The MouseMotionListener Interface

The corresponding adapter classes are`MouseMotionAdapter` *and* `MouseAdapter`.

Method	Purpose
mouseDragged(MouseEvent)	Called in response to the user moving the mouse while holding a mouse button down. This event is fired by the component that fired the most recent mouse-pressed event, even if the cursor is no longer over that component.
mouseMoved(MouseEvent)	Called in response to the user moving the mouse with no mouse buttons pressed. This event is fired by the component that's currently under the cursor.

The MouseEvent Class

Method	Purpose
int getClickCount()	Returns the number of quick, consecutive clicks the user has made (including this event). For example, returns 2 for a double click.
int getX() int getY() Point getPoint()	Return the (x,y) position at which the event occurred, relative to the component that fired the event.

7)Window Listeners

When the appropriate listener has been registered on a window (such as a frame or dialog), window events are fired just after the window activity or state has occurred. A window is considered as a "focus owner", if this window receives keyboard input.

The following window activities or states can precede a window event:

- Opening a window — Showing a window for the first time.

- Closing a window — Removing the window from the screen.
- Iconifying a window — Reducing the window to an icon on the desktop.
- Deiconifying a window — Restoring the window to its original size.
- Focused window — The window which contains the "focus owner".
- Activated window (frame or dialog) — This window is either the focused window, or owns the focused window.
- Deactivated window — This window has lost the focus. For more information about focus, see theAWT Focus Subsystem specification.
- Maximizing the window — Increasing a window's size to the maximum allowable size, either in the vertical direction, the horizontal direction, or both directions.

The `WindowListener` interface defines methods that handle most window events, such as the events for opening and closing the window, activation and deactivation of the window, and iconification and deiconification of the window.

The WindowListener Interface

Method	Purpose
windowOpened(WindowEvent)	Called just after the listened-to window has been shown for the first time.
windowClosing(WindowEvent)	Called in response to a user request for the listened-to window to be closed. To actually close the window, the listener should invoke the window's `dispose`or`setVisible(false)`method.
windowClosed(WindowEvent)	Called just after the listened-to window has closed.
windowIconified(WindowEvent) windowDeiconified(WindowEvent)	Called just after the listened-to window is iconified or deiconified, respectively.
windowActivated(WindowEvent) windowDeactivated(WindowEvent) prefer the	Called just after the listened-to window is activated or deactivated, respectively. These methods are not sent to windows that are not frames or dialogs. For this reason, we 1.4`windowGainedFocus`and`windowLostFocus`methods

	to determine when a window gains or loses the focus.

The WindowEvent Class

Method	Purpose
Window getWindow()	Returns the window that fired the event. You can use this instead of the`getSource` method.

8) Key Listener

Key events indicate when the user is typing at the keyboard. Specifically, key events are fired by the component with the keyboard focus when the user presses or releases keyboard keys.

Notifications are sent about two basic kinds of key events:

- The typing of a Unicode character
- The pressing or releasing of a key on the keyboard

The first kind of event is called a *key-typed* event. The second kind is either a *key-pressed* or *key-released* event.

In general, you react to only key-typed events unless you need to know when the user presses keys that do not correspond to characters. For example, to know when the user types a Unicode character — whether by pressing one key such as 'a' or by pressing several keys in sequence — you handle key-typed events. On the other hand, to know when the user presses the F1 key, or whether the user pressed the '3' key on the number pad, you handle key-pressed events.

The KeyListener Interface

The corresponding adapter class is `KeyAdapter`.

Method	Purpose
keyTyped(KeyEvent)	Called just after the user types a Unicode character into the

	listened-to component.
keyPressed(KeyEvent)	Called just after the user presses a key while the listened-to component has the focus.

keyReleased(KeyEvent)	Called just after the user releases a key while the listened-to component has the focus.

The KeyEvent Class

The `KeyEvent` class inherits many useful methods from the`InputEvent` class, such as `getModifiersEx`, and a couple of useful methods from the `ComponentEvent` and`AWTEvent` classes. See the InputEvent Class table in themouse listener page for a complete list.

Method	Purpose
int getKeyChar()	Obtains the Unicode character associated with this event. Only rely on this value for key-typed events.
int getKeyCode()	Obtains the key code associated with this event. The key code identifies the particular key on the keyboard that the user pressed or released. The `KeyEvent` class defines many key code constants for commonly seen keys. For example, `VK_A`specifies the key labeled **A**, and`VK_ESCAPE` specifies the Escape key.
String getKeyText(int) String getKeyModifiersText(int)	Return text descriptions of the event's key code and modifier keys, respectively.
boolean isActionKey()	Returns true if the key firing the event is an action key. Examples of action keys include Cut, Copy, Paste, Page Up, Caps Lock, the arrow and function keys. This information is valid only for key-pressed and key-released events.