OOPS Study Material UNIT-5

Object Oriented Programming (Anna University)

UNIT V JAVAFX EVENT HANDLING, CONTROLS AND COMPONENTS

JAVAFX Events and Controls: Event Basics – Handling Key and Mouse Events. Controls: Checkbox, ToggleButton – RadioButtons – ListView – ComboBox – ChoiceBox – Text Controls – ScrollPane. Layouts – FlowPane – HBox and VBox – BorderPane – StackPane – GridPane. Menus – Basics – Menu – Menu bars – MenuItem.

BASICS OF EVENT HANDLING

Definition: Event

Changing the state of an object is known as an event. i.e. event describes the change in state of source. Events are generated as result of user interaction with the graphical user interface components. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen.

What is Event Handling?

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism has the code which is known as event handler that is executed when an event occurs.

Events are included in the following packages:

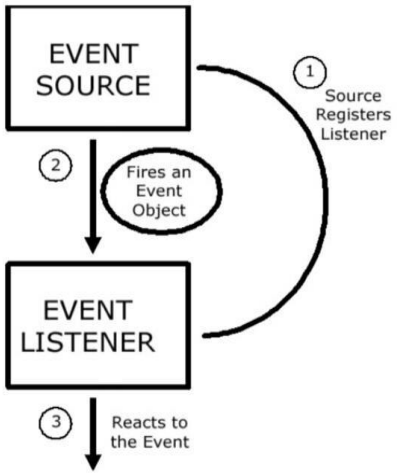
1) java.util
2) java.awt
3) java.awt.event

DELEGATION EVENT MODEL

Delegation Event Model is the modern approach to handle the event. It defines the standard and consistent mechanism to generate and process events.

Concept:

A source generates an event and sends it to one or more listeners.
Listener simply waits until it receives an event.
Once received, the listener processes the event and then returns.
User interface element is able to "delegate" the processing of an event to a separate piece of code.
Notifications are sent only to those listeners that want to receive them.



- The Delegation Event Model is based on the concept of **"Event source"** and **"Event Listeners"**
- Any object that is interested in receiving messages (or events) is called an **Event Listener.**
- Any object that generates the messages (or Events) is called an **Event Source.**

COMPONENTS OF DELEGATION EVENT MODEL:

There are three major components in the delegation event model:

1. Events
2. Event Sources
3. Event Listeners

1. Events

An event is an object that describes a state change in a source. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.

2. Event Sources

A Event Source is an object that generates an event. Sources may generate more than one type of event. A source must register listeners in order for the listeners to receive notifications about a specific type of event.

3. Event Listeners

Listener is an object that is notified when an event occurs. It has two major requirements:

1. It must have been registered with one or more sources to receive notifications about specific types of events.
2. It must implement methods to receive and process these notifications.

Event	Event Source	Description	Event Listener
ActionEvent	Button	Generates action events when the button is pressed.	ActionListener
ItemEvent	Checkbox	Generates item events when the check box is selected or deselected.	ItemListener
ItemEvent	Choice	Generates item events when the choice is changed.	ItemListener
ItemEvent	List	Generates action events when an item is double-clicked;	ItemListener
MouseEvent	Mouse	Generates Mouse events when Mouse input occurs.	MouseListener
KeyEvent	Keyboard	Generates Key events when keyboard input occurs.	KeyListener

The package java.awt.event defines several types of events that are generated by various user interface elements.

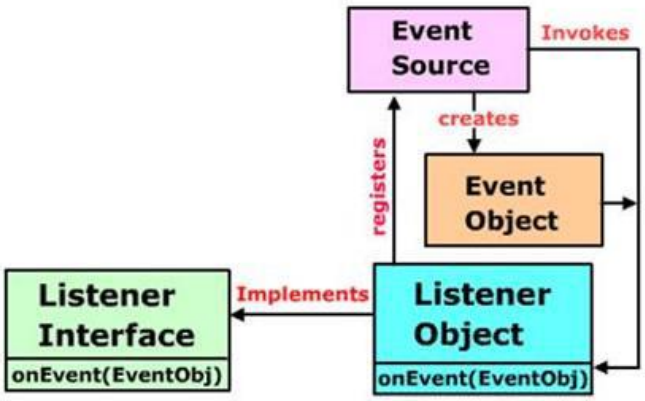
WORKING OF EVENT HANDLING

The following steps give an overview of how event handling in the AWT works:

1. A listener object is an instance of a class that implements a special interface called (naturally enough) a **listener interface.**
2. An event source is an object that can register listener objects and send them event objects.

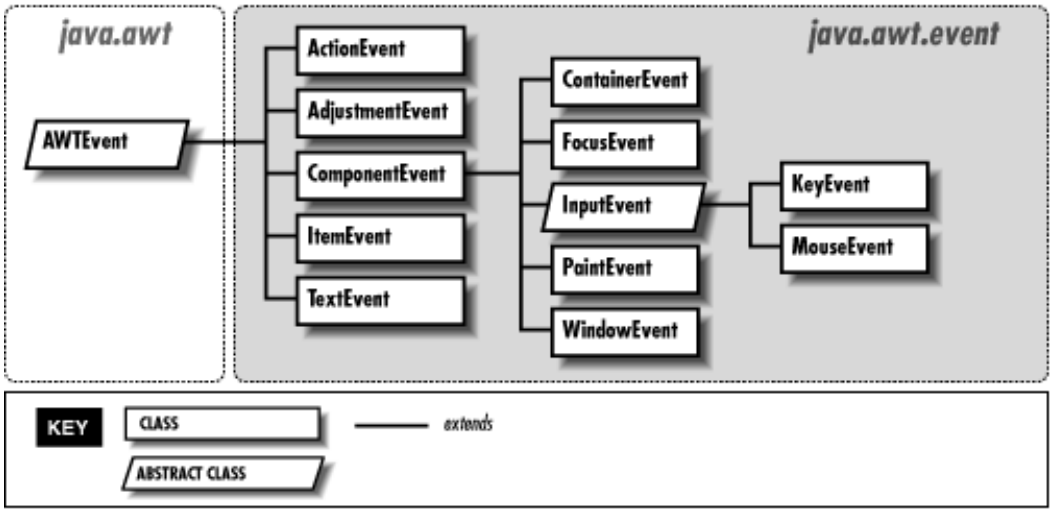
3. The event source sends out event objects to all registered listeners when that event occurs.
4. The listener objects will then use the information in the event object to determine their reaction to the event.

You register the listener object with the source object by using lines of code that follow the model:
eventSourceObject.addEventListener(eventListenerObject);
Example:
Button b1=new Button(―OK‖);
B1.addActionListener(this);
The listener object is notified whenever an ―action event‖ occurs on the button (when the button is clicked).
The below figure explains the working of Delegation Event Model:



Advantages of Event Delegation Model:
1. In event delegation model, the events are handled using objects. This allows a clear separation between the usage of the components and the design.
2. It accelerates the performance of the application in which multiple events are used.

AWT EVENT HIERARCHY
 Event handling in Java is object oriented, with all events descending from EventObject class in the java.util.package.
 The EventObject class has a subclass AWTEvent, which is the parent of all AWT event classes.
 The following diagram shows the hierarchy of AWT event class:



AWT Event Classes:

- AWT Event is the subclass of EventObject class.
- The subclasses of AWT Event class can be categorized into two:

1. Semantic Events
2. Low-level Events

1. Semantic Events:

A semantic event is one that expresses what the user is doing, such as ―clicking the button‖. The following event classes are semantic event classes:

1. ActionEvent
2. AdjustmentEvent
3. ItemEvent
4. TextEvent

2. Low-level Events:

Low-level events are those that makes the semantic events possible. For example, a semantic event ―button click‖ involves series of low level events such as mouse down, mouse moves and a mouse up. The following event classes are Low-level event classes:

1. ComponentEvent
2. ContainerEvent
3. FocusEvent
4. KeyEvent
5. MouseEvent
6. WindowEvent

The following table shows the description of various event classes:

Event classes	Description	Event Source
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.	Button, JButton
AdjustmentEvent	Generated when a scrollbar is manipulated	Scrollbar, JScrollbar

ComponentEvent	Generated when a component is hidden, moved, resized or becomes visible.	Component
ContainerEvent	Generated when a component is added to or removed from a container	Component
FocusEvent	Generated when a component gains or loses keyboard focus	Component
ItemEvent	Generated when a checkbox or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.	List, JList Choice, Checkbox
KeyEvent	Generated when the input is received from the keyboard	keyboard
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed or released; also generated when the mouse enters or exits a component.	Mouse
MouseWheelEvent	Generated when the mouse wheel is moved	Mouse
TextEvent	Generated when the value of the text field or text area is changed.	TextField or TextArea
WindowEvent	Generated when a window is activated, closed, deactivated.	Window

EVENT LISTENERS:

The task of handling an event is carried out by **Event Listeners**. When an event occurs,

1. An event object of the appropriate type is created.
2. This object is then passed to a Listener.
3. A listener must implement the interface that has the methods for event handling

Source	**Event Class**	**Class Methods**	**Listener Interface**	**Interface Methods**
Button	ActionEvent	String getActionCommand( )	ActionListener	actionPerformed(ActionEvent ae)
List, Choice, Checkbox	ItemEvent	Object getItem( ) ItemSelectable getItemSelectable( )	ItemListener	itemStateChanged(ItemEvent ie)
Keyboard	KeyEvent	char getKeyChar( ) int getKeyCode( )	KeyListener	keyPressed(KeyEvent ke) keyReleased(KeyEvent ke) keyTyped(KeyEvent ke)

Mouse	MouseEvent	int getX( ) int getY( )	MouseListener	mouseClicked(MouseEvent me) mouseEntered(MouseEvent me) mouseExited(MouseEvent me) mousePressed(MouseEvent me) mouseReleased(MouseEvent me)
MouseMotionListener		mouseDragged(MouseEvent me) mouseMoved(MouseEvent me)		
Scrollbar	AdjustmentEvent	Adjustable getAdjustable() int getAdjustmentType() int getValue()	AdjustmentListener	adjustmentValueChanged(AdjustmentEvent ae)
Component	FocusEvent	Boolean isTemporary()	FocusListener	focusGained(FocusEvent fe) focusLost(FocusEvent fe)
TextField and TextArea	TextEvent	--	TextListener	textValueChanged(TextEvent)
Window	WindowEvent	Window getWindow() int getOldState() int getNewState()	WindowListener	windowActivated(WindowEvent we) windowClosed(WindowEvent we) windowClosing(WindowEvent we) windowDeactivated(WindowEvent we) windowDeiconified(WindowEvent we) windowIconified(WindowEvent we) windowOpened(WindowEvent we)
Component	ComponentEvent	Component getComponent()	ComponentListener	componentHidden(ComponentEvent ce) componentMoved(ComponentEvent ce) componentResized(ComponentEvent ce) componentShown(ComponentEvent ce)
Container	Containe	Component getChild()	ContainerListener	componentAdded(Conta

	rEvent	Container getContainer()		inerEvent ce) componentRemoved(Co ntainerEvent ce)

Registering Event Listeners:
Steps:
1. Either create a class that **implements** a listener interface or **extend** a class that implements a listener interface.
Example:
public class MyClass implements **ActionListener {**

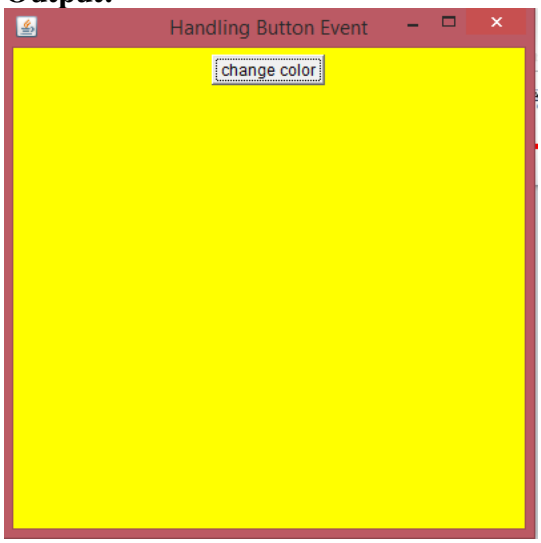
}
2. Register your listener with the source.
Example:
Component.addActionListener(instanceOfMyClass)
3. Implement the user actions by overriding the methods of listener interface
Example:
public void actionPerformed(ActionEvent e)
{

// code that reacts to the action or event
--
}
Example: Program to toggle the background color on every click of button

```
import java.awt.*;
import java.awt.event.*;
/* class implementing ActionListener interface and it must
override all the methods of the listener interface */
public class ToggleButton extends Frame implements ActionListener
{
boolean flag=true;
Button b1;
ToggleButton(String s)
{
super(s);
setSize(400,400);
setVisible(true);
setLayout(new FlowLayout());
b1=new Button("change color");
add(b1); // placing the button control
b1.addActionListener(this); // b1=event source, registering event listener
}
public void actionPerformed(ActionEvent ae) // code to handle the event
{
String str=ae.getActionCommand();
```

```
if(str.equals("change color"))
{
flag=!flag;
repaint();
}
}
public void paint(Graphics g)
{
if(flag)
setBackground(Color.red);
else
setBackground(Color.yellow);
}
public static void main(String[] arg)
{
ToggleButton T=new ToggleButton("Handling Button Event");
}
}
```

Output:



HANDLIND MOUSE, KEYBOARD AND WINDOW EVENTS

HANDLING MOUSE EVENTS

 Mouse events are generated when the mouse is dragged, moved, clicked, pressed or released; also generated when the mouse enters or exits a component.

 To handle Mouse events, class must implement the **MouseListener** & **MouseMotionListener** interface. Register mouse listener & mouse motion listener to receive notifications about MouseEvents.

Syntax:

```
addMouseListener(this);
addMouseMotionListener(this);
```

Description:

Source: **Mouse Event**

Class: **java.awt.event.MouseEvent**
Listener Interface: **java.awt.event.MouseListener**
java.awt.event.MouseMotionListener
Example:
The following program demonstrates Mouse event handling. When user drag the mouse it draws a line along the motion path.

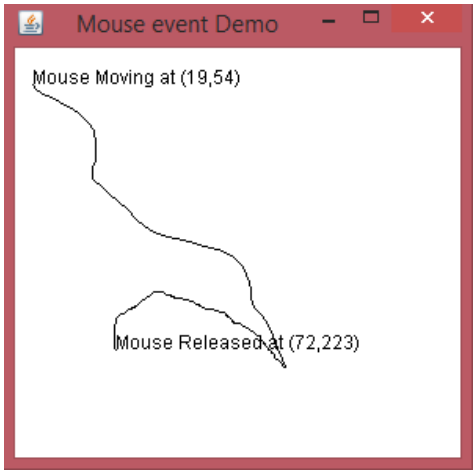
```
import java.awt.*;
import java.awt.event.*;
public class MouseHandler extends Frame implements MouseListener,MouseMotionListener
{
int x1,y1,x2,y2;
String str;
MouseHandler(String s)
{
super(s);
setSize(300,300);
setVisible(true);
addMouseListener(this);
addMouseMotionListener(this);
}
public void mouseDragged(MouseEvent me)
{
x1=x2;y1=y2;
x2=me.getX();
y2=me.getY();
Graphics g=this.getGraphics();
g.drawLine(x1,y1,x2,y2);
}
public void mouseMoved(MouseEvent me)
{
x1=x2;y1=y2;
x2=me.getX();
y2=me.getY();
str="Mouse Moving at ("+x2+","+y2+")";
repaint();
}
public void mouseClicked(MouseEvent me)
{
x2=me.getX();
y2=me.getY();
str="Mouse Clicked at ("+x2+","+y2+")";
repaint();
```

```
}
    public void mouseEntered(MouseEvent me)
    {
    x2=200;y2=100;
    str="Mouse Entered";
    repaint();
    }
    public void mouseExited(MouseEvent me)
    {
    x2=200;y2=100;
    str="Mouse Exited";
    repaint();
    }
    public void mousePressed(MouseEvent me)
    {
    x1=x2=me.getX();
    x1=y2=me.getY();
    }
    public void mouseReleased(MouseEvent me)
    {
    str="Mouse Released at ("+x2+","+y2+")";
    Graphics g=this.getGraphics();
    g.drawString(str,x2,y2);
    }
    public void paint(Graphics g)
    {
    g.drawString(str,x2,y2);
    }
    public static void main(String arg[])
    {
    MouseHandler ob=new MouseHandler("Mouse event Demo");
    }
    }
```

Program Explanation:

In the above program, the **MouseHanlder** class extends **Frame** and implements both **MouseListener** and **MouseMotionListener** interfaces to handle mouse events. These two interfaces contain methods to receive and process the various types of mouse events. Here, ―me‖ is a reference to the object receiving mouse events. The Frame then implements all the methods defined by the **MouseListener** and **MouseMotionListener** interfaces. These are the event handlers for the various mouse events. Each method handles its event and then returns.

Output:



HANDLING KEYBOARD EVENTS

 Keyboard events are generated when the input is received from the keyboard

 To handle keyboard events, class must implement the **KeyListener** interface. Register key listener to receive notifications about KeyEvents.

Syntax:

addKeyListener(this);

Description:

Source: **KeyBoard**

Event Class: **java.awt.event.KeyEvent**

Listener Interface: **java.awt.event.KeyListener**

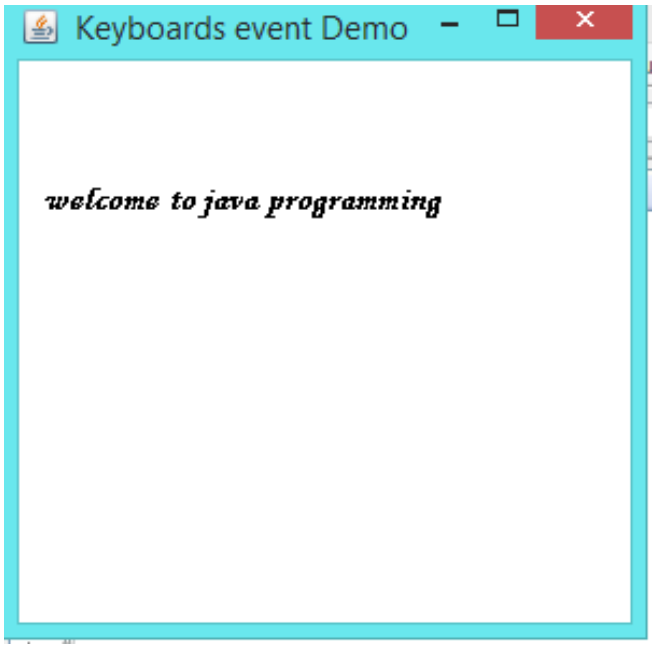
Example:

The following program demonstrates keyboard input. When program receives keystrokes, identifies the key and perform the corresponding actions specified by the program.

```
import java.awt.*;
import java.awt.event.*;
public class KeyboardHandler extends Frame implements KeyListener
{
int x=20,y=20;
String msg="";
KeyboardHandler(String s)
{
super(s);
setSize(300,300);
setVisible(true);
addKeyListener(this);
requestFocus();
}
public void keyPressed(KeyEvent ke)
{
Font f=new Font("Monotype Corsiva",Font.BOLD,15);
msg+=ke.getKeyChar();
```

```
setFont(f);
}
public void keyTyped(KeyEvent ke){ }
public void keyReleased(KeyEvent ke)
{
repaint();
}
public void paint(Graphics g)
{
g.drawString(msg,20,100);
}
public static void main(String arg[])
{
KeyboardHandler ob=new KeyboardHandler("Keyboard event Demo");
}
}
```

Output:



Program Explanation:

In the above program, the class extends **Frame** class and implements **KeyListener** to handle the event generated through keyboard. When a key is pressed, a **KEY_PRESSED** event is generated. This results in a call to the **keyPressed()** event handler. This handler gets the key typed by the user through **getKeyChar() method** and collects the character in the string variable **msg.** When the key is released, a **KEY_RELEASED** event is generated. The keyReleased() event handler calls the **repaint()** method to display the message on the frame window.

INTRODUCTION OF SWING

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

JApplet

Fundamental to Swing is the **JApplet** class, which extends **Applet**. Applets that useSwing must be subclasses of **JApplet**. **JApplet** is rich with functionality that is notfound in **Applet**. For example, **JApplet** supports various ‑panes,‖ such as the content pane, the glass pane, and the root pane. For the examples in this chapter, we will not be using most of **JApplet**'s enhanced features. However, one difference between **Applet** and **JApplet** is important to this discussion, because it is used by the sample applets in this chapter. When adding a component to an instance of **JApplet**, do not invoke the **add( )** method of the applet. Instead, call **add( )** for the *content pane* of the **JApplet** object. The content pane can be obtained via the method shown here:

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

Here, *comp* is the component to be added to the content pane.Icons and Labels
In Swing, icons are encapsulated by the **ImageIcon** class, which paints an icon from an image.Two of its constructors are shown here:

ImageIcon(String *filename*)ImageIcon(URL *url*)

The first form uses the image in the file named *filename*. The second form uses the image in theresource identified by *url*.
The **ImageIcon** class implements the **Icon** interface that declares the methodsshown here:
Method Description
int getIconHeight( ) Returns the height of the iconin pixels.
int getIconWidth( ) Returns the width of the iconin pixels.
void paintIcon(Component *comp*, Graphics *g*,int *x*, int *y*)
Paints the icon at position *x*, *y* on the graphics context *g*. Additional information about the paint operation can be provided in *comp*.
Swing labels are instances of the **JLabel** class, which extends **JComponent**. It can display textand/or an icon. Some of its constructors are shown here:
JLabel(Icon *i*) Label(String *s*)
JLabel(String *s*, Icon *i*, int *align*)

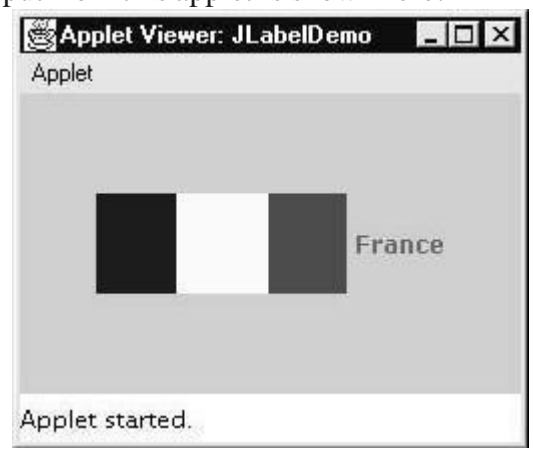
Here, *s* and *i* are the text and icon used for the label. The *align* argument is either **LEFT**,**RIGHT**, **CENTER**, **LEADING**, or **TRAILING**. These constants are defined in the **SwingConstants** interface, along with several others used by the Swing classes.

The icon and text associated with the label can be read and written by thefollowing methods:
Icon getIcon( ) String getText( ) void setIcon(Icon *i*)
void setText(String *s*)
Here, *i* and *s* are the icon and text, respectively.
The following example illustrates how to create and display a label containing bothan icon and a string. The applet begins by getting its content pane. Next, an **ImageIcon** object is created for the file **france.gif**.

This is used as the second argument to the **JLabel** constructor. The first and last arguments for the **JLabel** constructor are the label text and the alignment. Finally, the labelis added to the content pane.

```
import java.awt.*; import javax.swing.*;
/*
<applet code="JLabelDemo" width=250 height=150>
</applet>
*/
public class JLabelDemo extends JApplet {public void init() {
// Get content pane
Container contentPane = getContentPane();
// Create an icon
ImageIcon ii = new ImageIcon("france.gif");
// Create a label
JLabel jl = new JLabel("France", ii, JLabel.CENTER);
// Add label to the content panecontentPane.add(jl);
}
}
```

Output from this applet is shown here:



Text Fields

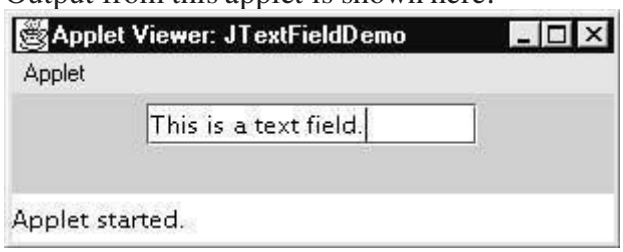
The Swing text field is encapsulated by the **JTextComponent** class, which extends **JComponent**. It provides functionality that is common to Swing text components. One of its subclasses is **JTextField**, which allows you to edit one line of text. Some of its constructors are shown here:

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

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

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

Output from this applet is shown here:



Buttons

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

The following are the methods that control this behavior:
void setDisabledIcon(Icon *di*) void setPressedIcon(Icon *pi*) void setSelectedIcon(Icon *si*) void setRolloverIcon(Icon *ri*)

Here, *di*, *pi*, *si*, and *ri* are the icons to be used for these different conditions.
The text associated with a button can be read and written via the following methods:
String getText( )
void setText(String *s*)

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

Concrete subclasses of **AbstractButton** generate action events when they are pressed. Listenersregister and unregister for these events via the methods shown here:
void addActionListener(ActionListener *al*) void removeActionListener(ActionListener *al*)Here, *al* is the action listener.

AbstractButton is a superclass for push buttons, check boxes, and radio buttons.Each is examined next.

The JButton Class

The **JButton** class provides the functionality of a push button. **JButton** allows an icon,a string,or both to be associated with the push button. Some of its constructors are shown here: JButton(Icon *i*)
JButton(String *s*) JButton(String *s*, Icon *i*)
Here, *s* and *i* are the string and icon used for the button.

Check Boxes

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

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

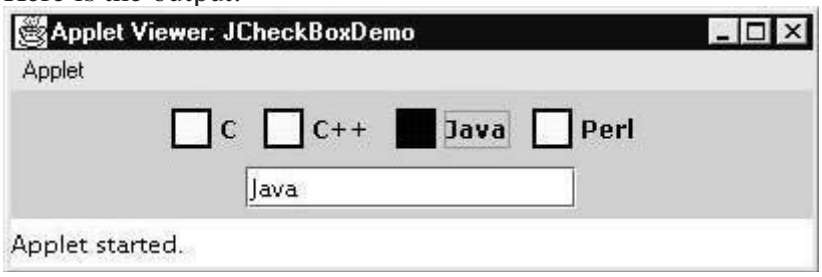
Here, *i* is the icon for the button. The text is specified by *s*. If *state* is **true**, the check box isinitially selected. Otherwise, it is not.
The state of the check box can be changed via the following method:void setSelected(boolean *state*)
Here, *state* is **true** if the check box should be checked.
The following example illustrates how to create an applet that displays four checkboxesand a text field. When a check box is pressed, its text is displayed in the text field.
The content pane for the **JApplet** object is obtained, and a flow layout is assigned as itslayout manager. Next, four check boxes are added to the content pane, and icons are assigned for the normal, rollover, and selected states. The applet is then registered to receive item events. Finally,a text field is added to the content pane.

When a check box is selected or deselected, an item event is generated. This is handled by **itemStateChanged( )**. Inside **itemStateChanged( )**, the **getItem( )** method gets the **JCheckBox** object that generated the event. The **getText( )** method gets the text for that check box and uses itto set the text inside the text field.

```
import java.awt.*; i
mport java.awt.event.*;
import javax.swing.*;
/*
<applet code="JCheckBoxDemo" width=400 height=50>
</applet>
*/
public class JCheckBoxDemo extends JAppletimplements ItemListener {
JTextField jtf; public void init() {
// Get content pane
Container contentPane = getContentPane();contentPane.setLayout(new FlowLayout());
// Create icons
```

```
ImageIcon normal = new ImageIcon("normal.gif"); ImageIcon rollover = new ImageIcon("rollover.gif");
ImageIcon selected = new ImageIcon("selected.gif");
// Add check boxes to the content pane JCheckBox cb = new JCheckBox("C", normal);
cb.setRolloverIcon(rollover); cb.setSelectedIcon(selected); cb.addItemListener(this);
contentPane.add(cb);
cb = new JCheckBox("C++", normal); cb.setRolloverIcon(rollover); cb.setSelectedIcon(selected);
cb.addItemListener(this); contentPane.add(cb);
cb = new JCheckBox("Java", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
cb = new JCheckBox("Perl", normal);cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
// Add text field to the content pane
jtf = new JTextField(15); contentPane.add(jtf);
}
public void itemStateChanged(ItemEvent ie)
{
JCheckBox cb = (JCheckBox)ie.getItem(); jtf.setText(cb.getText());
}
}
```

Here is the output:



Radio Buttons

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

JRadioButton(Icon *i*) JRadioButton(Icon *i*, boolean *state*) JRadioButton(String *s*) JRadioButton(String *s*, boolean *state*)JRadioButton(String *s*, Icon *i*)
JRadioButton(String *s*, Icon *i*, boolean *state*)
Here, *i* is the icon for the button. The text is specified by *s*. If *state* is **true**, the button isinitially selected. Otherwise, it is not.

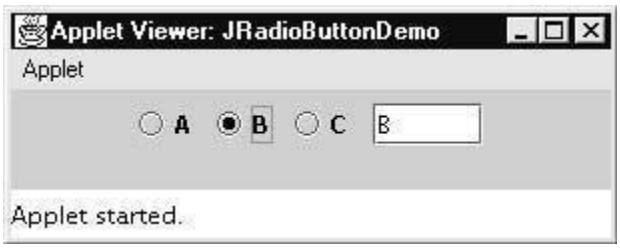
Radio buttons must be configured into a group. Only one of the buttons in that group can be selected at

any time. For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected. The **ButtonGroup** class is instantiated to create a button group. Its default constructor is invokedfor this purpose. Elements are then added to the button group via the following method:
void add(AbstractButton *ab*)
Here, *ab* is a reference to the button to be added to the group.

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

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JRadioButtonDemo" width=300 height=50>
</applet>
*/
public class JRadioButtonDemo extends JAppletimplements ActionListener {
JTextField tf; public void init() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Add radio buttons to content pane
JRadioButton b1 = new JRadioButton("A");
b1.addActionListener(this);
contentPane.add(b1);
JRadioButton b2 = new JRadioButton("B");
b2.addActionListener(this); contentPane.add(b2);
JRadioButton b3 = new JRadioButton("C");
b3.addActionListener(this);
contentPane.add(b3);
// Define a button group
ButtonGroup bg = new ButtonGroup();
bg.add(b1);
bg.add(b2);
bg.add(b3);
// Create a text field and add it
// to the content pane
tf = new JTextField(5);
contentPane.add(tf);
}
public void actionPerformed(ActionEvent ae)
{
tf.setText(ae.getActionCommand());
```

}
}
Output from this applet is shown here:



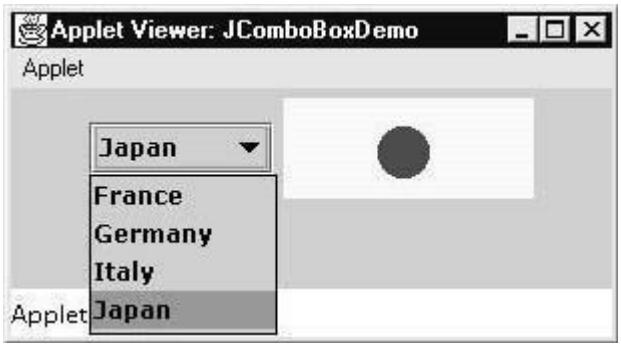
Combo Boxes

Swing provides a *combo box* (a combination of a text field and a drop-down list) through the **JComboBox** class, which extends **JComponent**. A combo box normally displays one entry. However, it can also display a drop-down list that allows a user to select a different entry. 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 isshown here:
void addItem(Object *obj*)
Here, *obj* is the object to be added to the combo box.
The following example contains a combo box and a label. The label displays an icon. The combobox contains entries for "France", "Germany", "Italy", and "Japan".
When a country is selected, the label is updated to display the flag for that country.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JComboBoxDemo" width=300 height=100>
</applet>
*/
public class JComboBoxDemo extends JAppletimplements ItemListener {
JLabel jl;
ImageIcon france, germany, italy, japan;
public void init() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Create a combo box and add it
// to the panel
JComboBox jc = new JComboBox();
jc.addItem("France");
jc.addItem("Germany");
jc.addItem("Italy");
 jc.addItem("Japan");
jc.addItemListener(this);
```

```
contentPane.add(jc);
// Create label
jl = new JLabel(new ImageIcon("france.gif"));
contentPane.add(jl);
}
public void itemStateChanged(ItemEvent ie)
{
String s = (String)ie.getItem();
jl.setIcon(new ImageIcon(s + ".gif"));
}
}
```

Output from this applet is shown here:



Tabbed Panes

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

Tabbed panes are encapsulated by the **JTabbedPane** class, which extends **JComponent**. We will use its default constructor. Tabs are defined via the following method:

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

Here, *str* is the title for the tab, and *comp* is the component that should be added to thetab. Typically, a **JPanel** or a subclass of it is added.

The general procedure to use a tabbed pane in an applet is outlined here:

Create a **JTabbedPane** object.

Call **addTab( )** to add a tab to the pane. (The arguments to this method definethe title of the tab and the component it contains.)

Repeat step 2 for each tab.

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

Scroll Panes

A *scroll pane* is a component that presents a rectangular area in which a component may beviewed. Horizontal and/or vertical scroll bars may be provided if necessary.

Scroll panes are implemented in Swing by the **JScrollPane** class, which extends**JComponent**.Some of its constructors are shown here:

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

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

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

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

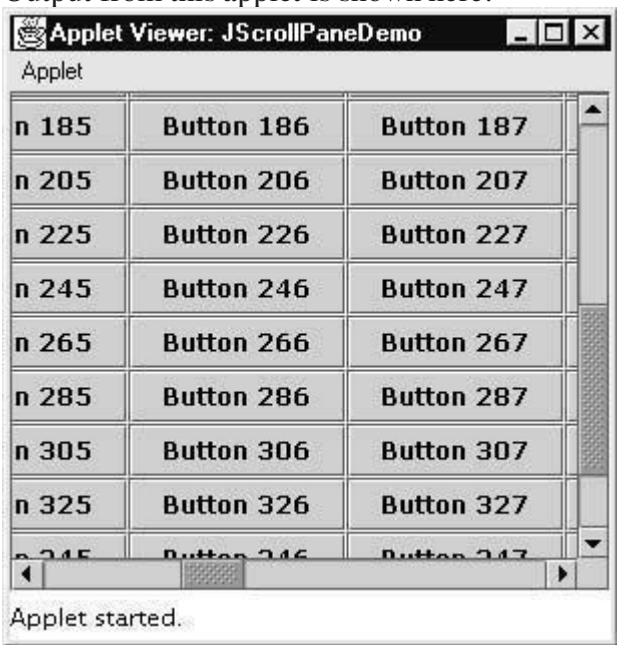
Here are the steps that you should follow to use a scroll pane in an applet:
Create a **JComponent** object.
Create a **JScrollPane** object. (The arguments to the constructor specify thecomponent and the policies for vertical and horizontal scroll bars.)
Add the scroll pane to the content pane of the applet.

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

```
 import java.awt.*;
import javax.swing.*;
/*
<applet code="JScrollPaneDemo" width=300 height=250>
</applet>
*/
public class JScrollPaneDemo extends JApplet
{
public void init()
{
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new BorderLayout());
// Add 400 buttons to a panel
JPanel jp = new JPanel();
jp.setLayout(new GridLayout(20, 20));
int b = 0;
for(int i = 0; i < 20; i++)
{
for(int j = 0; j < 20; j++)
{
jp.add(new JButton("Button " + b));
++b;
}
}
```

```
// Add panel to a scroll pane
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane jsp = new JScrollPane(jp, v, h);
// Add scroll pane to the content pane contentPane.add(jsp, BorderLayout.CENTER);
}
}
```

Output from this applet is shown here:



Trees

A *tree* is a component that presents a hierarchical view of data. A user has the ability to expandor 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 ofthe 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 thetree 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.

Tables

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

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:

Create a **JTable** object.

Create a **JScrollPane** object. (The arguments to the constructor specify the tableand the policies for vertical and horizontal scroll bars.)

Add the table to the scroll pane.

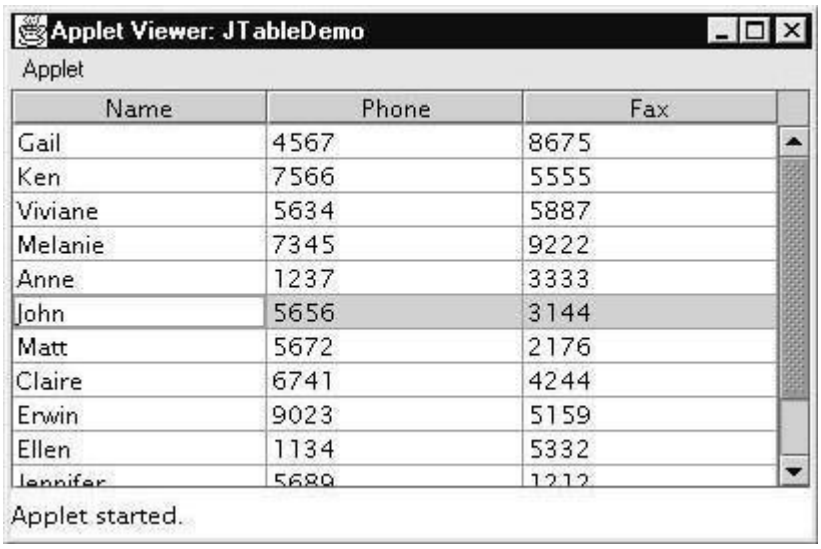
Add the scroll pane to the content pane of the applet.

The following example illustrates how to create and use a table. The content pane of the **JApplet**object is obtained and a border layout is assigned as its layout manager.A one-dimensional array of strings is created for the column headings. This table has three columns. A two-dimensional array of strings is created for the table cells. You can see that each element in the array is an array of three strings. These arrays are passed to the **JTable** constructor. The table is added to a scroll pane and then the scroll pane is added to the content pane.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTableDemo" width=400 height=200>
</applet>
*/
public class JTableDemo extends JApplet {

public void init() {
// Get content pane
Container contentPane = getContentPane();
// Set layout manager contentPane.setLayout(new BorderLayout());
// Initialize column headings
final String[] colHeads = { "Name", "Phone", "Fax" };
// Initialize data
final Object[][] data = {
{ "Gail", "4567", "8675" },
{ "Ken", "7566", "5555" },
{ "Viviane", "5634", "5887" },
{ "Melanie", "7345", "9222" },
{ "Anne", "1237", "3333" },
{ "John", "5656", "3144" },
```

```
{ "Matt", "5672", "2176" },
{ "Claire", "6741", "4244" },
{ "Erwin", "9023", "5159" },
{ "Ellen", "1134", "5332" },
{ "Jennifer", "5689", "1212" },
{ "Ed", "9030", "1313" },
{ "Helen", "6751", "1415" }
};
// Create the table
JTable table = new JTable(data, colHeads);
// Add table to a scroll pane
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED; JScrollPane jsp = new JScrollPane(table, v, h);
// Add scroll pane to the content pane contentPane.add(jsp, BorderLayout.CENTER);
}
}
```

Here is the output:



Definition: Layout Management

Layout Management is the process of arranging components within a window. Layout manager automatically arranges several components within a window. Each container object has a layout manager associated with it.

- Panel,Applet - Flow Layout
- Frame - Border Layout

Whenever a container is resized, the layout manager is used to position each of the components within it.

General syntax for setting layout to container

void setLayout(LayouManager obj)

Various Layout Managers are

1. FlowLayout
2. BorderLayout
3. Grid Layout
4. GridbagLayout
5. Card Layout
6. BoxLayout

Arrange component without using layout Manager:

You can position components manually using setBounds() method defined by Component class.

1. Disable the default manager of your container.

setLayout(null);
2. Givethe location and size of the component which is to be added in the container. setBounds(int x, int y, int width, int height);
Example:
Button b=new Button(―click me‖);
b.setBounds(10,10,50,20);

FlowLayout

✓ FlowLayout arranges the components in rows from left-to-right and top-to-bottom order based on the order in which they were added to the container.

✓ FlowLayout arranges components in rows, and the alignment specifies the alignment of the rows. For example, if you create a FlowLayout that's left aligned, the components in each row will appear next to the left edge of the container.

✓ The flow layout manager lines the components horizontally until there is no more room and then starts a new row of components.

✓ When the user resizes the container, the layout manager automatically reflows the components to fill the available space.

Constructors:

· **FlowLayout()** - create default layout, which centers component and leaves 5 pixels spaces between each component.

· **FlowLayout(int how)-**specify how each line is aligned.

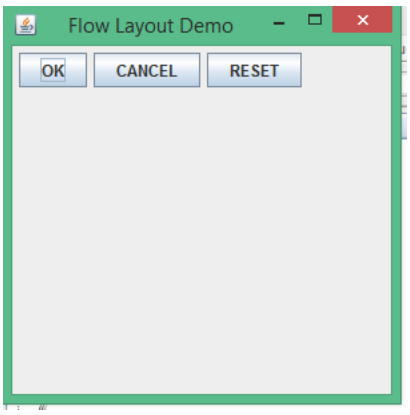
Valid values for **how** are:
FlowLayout.LEFT
FlowLayout.CENTER
FlowLayout.RIGHT

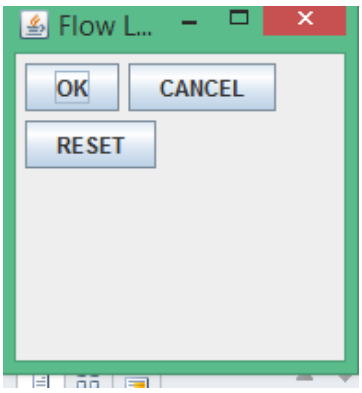
· **FlowLayout(int how, int horz, int vert)** - allow you to specify the horizontal and vertical gaps that should appear between components, and if you use a constructor that doesn't accept these values, they both default to 5.

Example:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class FlowDemo extends JFrame
{
 public FlowDemo()
 {
 setTitle("Flow Layout Demo");
 Container cp=getContentPane();
 cp.setLayout(new FlowLayout(FlowLayout.LEFT));
 JButton ok=new JButton("OK");
 JButton cancel=new JButton("CANCEL");
 JButton reset=new JButton("RESET");
 cp.add(ok);
 cp.add(cancel);
 cp.add(reset);
 setVisible(true);
 setSize(300,300);
 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 }
 public static void main(String[] args)
 {
 FlowDemo f=new FlowDemo();
 }
}
```

Output:





GridLayout

✓ The GridLayout layout manager divides the available space into a grid of cells, evenly allocating the space among all the cells in the grid and placing one component in each cell. ✓ Cells are always same size.
✓ When you resize the window, the cells grow and shrink, but all the cells have identical sizes.

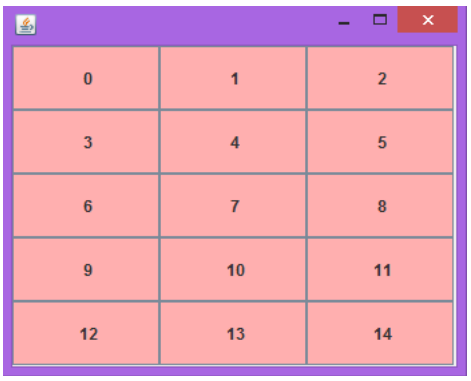
Constructors:

· **GridLayout(int rows, int cols**) - construct a grid with specified rows and cols. · **GridLayout(int rows, int cols, int hspace, int vspace)** - to specify the amount of horizontal and vertical space that should appear between adjacent components.

Example:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class GridDemo extends JFrame
{
 GridDemo()
 {
 JButton[] button=new JButton[15];
 int j=0;
 setSize(400,300);
 setVisible(true);
 Container cp=getContentPane();
 cp.setLayout(new GridLayout(5,5));
 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 for (int i=0;i<15; i++)
 {
 button[i]=new JButton(" "+i);
 button[i].setBackground(Color.pink);
 button[i].setFont(new Font("SanSerif", Font.BOLD,12));  cp.add(button[i]);
 }
}
public static void main(String[] arg)
{
 GridDemo cd=new GridDemo();
} }
```

Output:



Border Layout

✓ Border layout divides the container into five regions - North, West, East, South and Center.
✓ The five regions correspond to top, left, bottom, right and center of the container. ✓ Each region can have only one component.

Constructors:

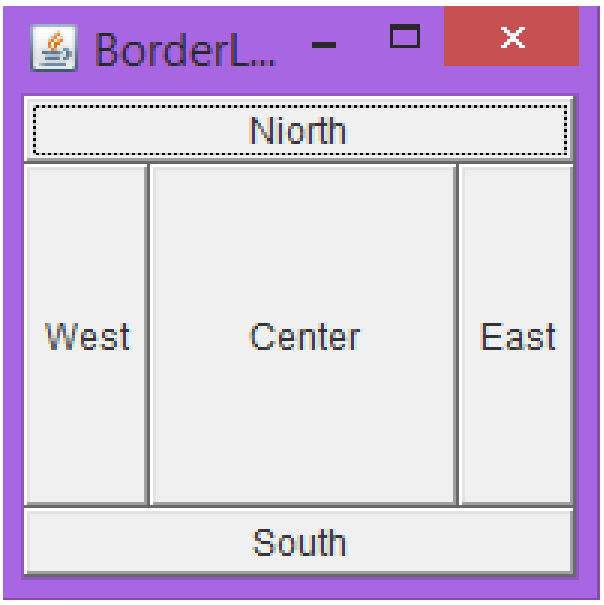
· BorderLayout()
· BorderLayout(int hspace, int vspace) – leave space between components.

Border layout grows all components to fill the available space.
You can add components by specifying a constraint as follows:
BorderLayout.CENTER|NORTH|SOUTH|EAST|WEST

Example:

```
import java.awt.*;
import javax.swing.*;
public class BorderDemo extends JFrame
{
 BorderLayout grid=new BorderLayout();
 Button b1=new Button("Niorth");
 Button b2=new Button("South");
 Button b3=new Button("Center");
 Button b4=new Button("East");
 Button b5=new Button("West");
 BorderDemo(String s)
 {
 super(s);
 setLayout(grid);
 add(b1,BorderLayout.NORTH);
 add(b2,BorderLayout.SOUTH);
 add(b3,BorderLayout.CENTER);
 add(b4,BorderLayout.EAST);
 add(b5,BorderLayout.WEST);
 setSize(200,200);
 setVisible(true);
 }
 public static void main(String arg[])
 {
 BorderDemo ob=new BorderDemo("BorderLayout Demo");
 }
}
```

Output:



GridBag Layout - Gridlayout without limitations

✓ In Grid bag layout, the rows and columns have variable sizes.
✓ It is possible to merge two adjacent cells and make a space for placing larger components.
✓ To describe the layout to grid bag manager, you must follow the procedure:
1. Create an object of type GridBagLayout. No need to specify rows and column. 2. Set this GridBagLayout object to the container.
3. Create an object of type GridBagConstraints. This object will specify how the components are laid out within the grid bag.
4. For each components, fill in the GridBagConstraints object.Finally add the component with the constraint by using the call:
add(Component, constraint);

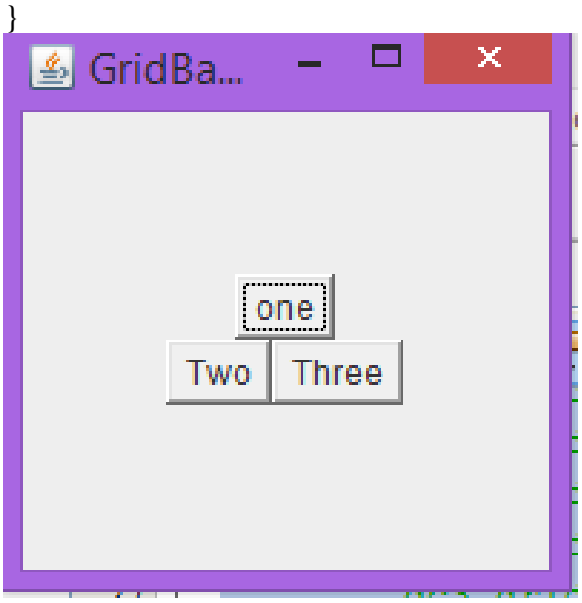
GridBagConstraints:

· Gridx – specify the column position of the component to be added
· Gridy - specify the row position of the component to be added
· Gridwidth- specify how many columns occupied by the component
· Gridheight - specify how many rows occupied by the component
· 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.
· anchor – used when the component is smaller than its display area to determine where to palce the component
· weightx – used to determine how to distribute space among columns, which is important for specifying resizing behaviour.
· weighty - used to determine how to distribute space among rows, which is important for specifying resizing behaviour.
· ipadx – specifies the component's internal padding within the layout, how much to add to the minimum width of the component. Default value=0.
· ipady - specifies the component's internal padding within the layout, how much to add to the minimum height of the component. Default value=0.
· insets – specifies the external padding of the component – the minimum amount of space between the component and the edges of its display are. Default is (0,0,0,0)

Example:

```
import java.awt.*;
import javax.swing.*;
public class GridBagDemo extends JFrame
{
 GridBagLayout gb=new GridBagLayout();
 GridBagConstraints gc1= new GridBagConstraints();
```

```
GridBagConstraints gc2= new GridBagConstraints();
GridBagConstraints gc3= new GridBagConstraints();
Button b1=new Button("one");
Button b2=new Button("Two");
Button b3=new Button("Three");
GridBagDemo(String s)
{
super(s);
setLayout(gb);
gc1.gridx=0;
gc1.gridy=0;
gc1.gridwidth=2;
gc1.gridheight=1;
gc2.gridx=0;
gc2.gridy=1;
gc2.gridwidth=1;
gc2.gridheight=1;
gc3.gridx=1;
gc3.gridy=1;
gc3.gridwidth=1;
gc3.gridheight=1;
add(b1,gc1);
add(b2,gc2);
add(b3,gc3);
setSize(200,200);
setVisible(true);
}
public static void main(String arg[])
{
GridBagDemo ob=new GridBagDemo("GridBagLayout Demo");
}
}
```



Card Layout

✓ The card layout is unique in which it stores several different layouts.

✓ Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time.

✓ You can prepare the other layouts and have them hidden, ready to be activated when needed.

Constructors:

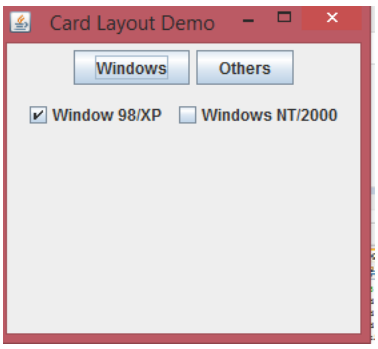
· **CardLayout() -** creates a default card layout.

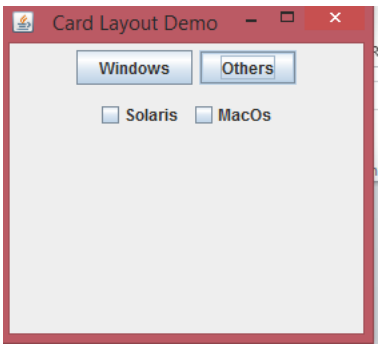
· **CardLayout(int horz, int vert)** - allows to specify the horizontal and vertical space left between components.

Example:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class CardDemo extends JFrame implements ActionListener
{
 JCheckBox win98, winNT, solaris, mac;
 JPanel osCards;
 CardLayout cardLO;
 JButton Win, Others;
 CardDemo(String s)
 {
 super(s);
 Win=new JButton("Windows");
 Others=new JButton("Others");
 add(Win);
 add(Others);
 cardLO=new CardLayout();
 osCards=new JPanel();
 osCards.setLayout(cardLO); // set Panel layout to card layout
 win98=new JCheckBox("Window 98/XP",null,true);
winNT=new JCheckBox("Windows NT/2000");
solaris=new JCheckBox("Solaris");
 mac=new JCheckBox("MacOs");
 JPanel winpan=new JPanel();
 winpan.add(win98);
 winpan.add(winNT);
 JPanel otherpan=new JPanel();
 otherpan.add(solaris);
 otherpan.add(mac);
 // add panels to card deck panel
 osCards.add(winpan,"Windows");
 osCards.add(otherpan,"Others");
 // add cards to main window JFrame
 add(osCards);
 Win.addActionListener(this);
 Others.addActionListener(this);
 setVisible(true);
 setSize(400,400);
 setLayout(new FlowLayout());
 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  }
 public void actionPerformed(ActionEvent ae)
 {
 if(ae.getSource()==Win)
 cardLO.show(osCards, "Windows");
 else
 cardLO.show(osCards, "Others");
 }
 public static void main(String[] args)
 {
 CardDemo obj=new CardDemo("Card Layout Demo");
 }
}
```

Output:





Box Layout

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

Constructor of BoxLayout class

BoxLayout(Container c, int axis): creates a box layout that arranges the components with the given axis.
The BoxLayout manager is designed with an axis parameter that specifies the type of layout. This can be done in four ways:

X_AXIS – components are placed horizontally from left to right
Y_AXIS - components are placed vertically from top to bottom
LINE_AXIS – components are placed in a line, based on the container's ComponentOrientation property

Table: LINE_AXIS

ComponentOrientation	Components
Horizontal	Components are kept vertically, otherwise kept horizontally
Horizontal; left to right	Placed left to right, otherwise right to left
Vertical Orientations	Laid from top to bottom

PAGE_AXIS – Components are placed the way text lines are written on a page, based on the container's ComponentOrientation property

Table: PAGE_AXIS

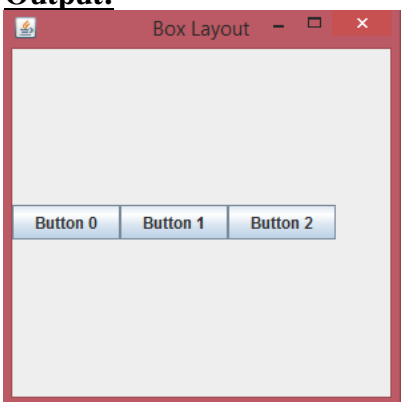
ComponentOrientation	Components
Horizontal	Horizontally, else vertically placed
Horizontal; left to right	Placed left to right, otherwise right to left
Vertical Orientations	Laid from top to bottom

Example:

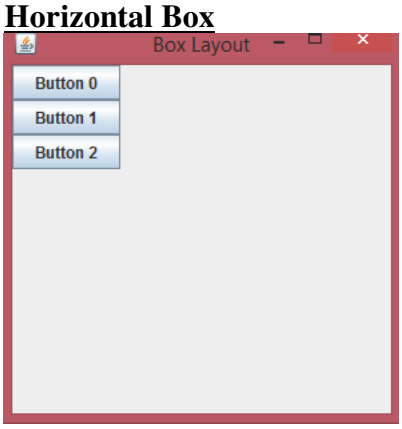
```
import java.awt.*;
import javax.swing.*;
public class BoxDemo extends JFrame
{
 void boxDemo()
 {
 setTitle("Box Layout");
```

```
Container content=getContentPane();
setVisible(true);
setSize(300,300);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
Box b1=new Box(BoxLayout.X_AXIS);
// Box b2=new Box(BoxLayout.Y_AXIS);
for(int i=0;i<3;i++)
{
b1.add(new JButton("Button "+i));
// b2.add(new JButton("Button "+i));
}
content.add(b1);
// content.add(b2);
}
public static void main(String[] args)
{
BoxDemo demo=new BoxDemo();
demo.boxDemo();
}
}
```

Output:



Horizontal Box



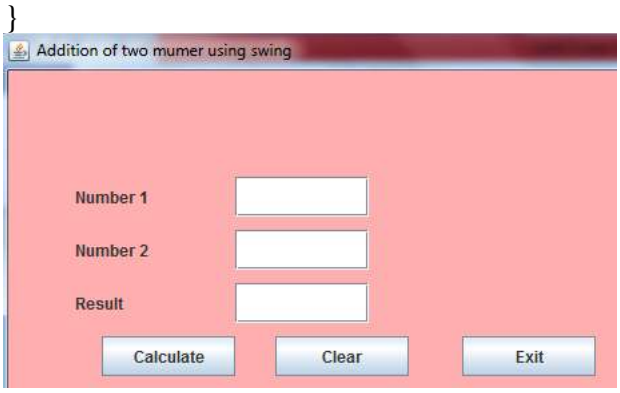
Vertical Box

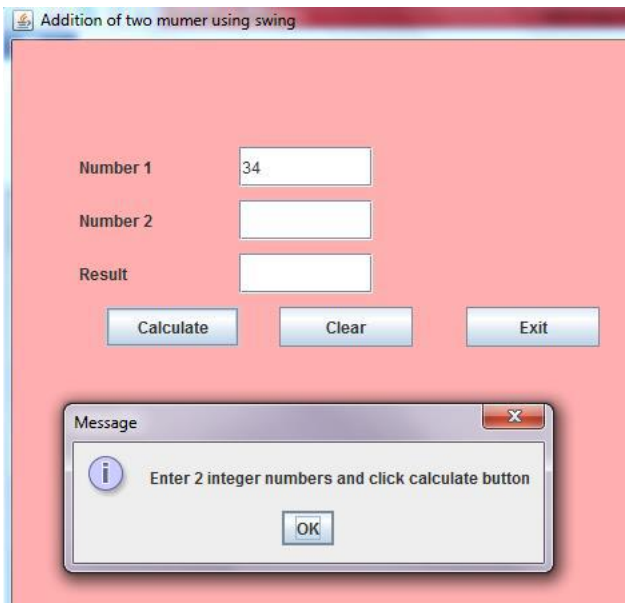
Example Program 1: ADDITION OF TWO NUMBERS

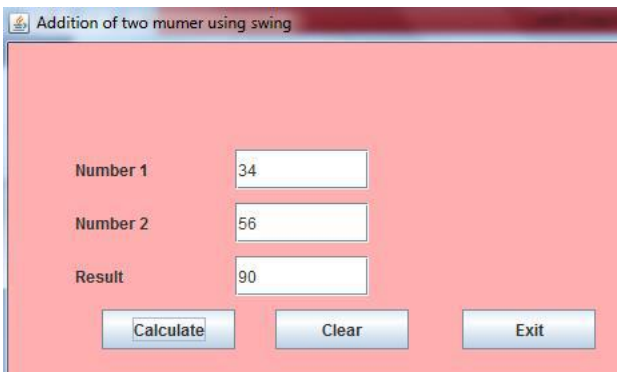
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class addition extends JFrame implements ActionListener { JFrame fr;
JLabel no1;
JLabel no2;
JLabel result;
JTextField F1;
JTextField F2;
JTextField F3;
JButton calculate;
JButton clear;
```

```
JButton exit;
public addition()
{
JFrame fr=new JFrame("Addition of two mumer using swing"); fr.setSize(1000,1000);
fr.setVisible(true);
fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
Container c=fr.getContentPane();
c.setLayout(null);
c.setBackground(Color.pink);
no1 = new JLabel("Number 1");
no1.setBounds(50,80,100,30);
F1 = new JTextField();
F1.setBounds(170,80,100,30);
no2 = new JLabel("Number 2");
no2.setBounds(50,120,100,30);
F2 = new JTextField();
F2.setBounds(170,120,100,30);
result = new JLabel("Result");
result.setBounds(50,160,100,30);
F3 = new JTextField();
F3.setBounds(170,160,100,30);
calculate=new JButton("Calculate");
calculate.setBounds(70,200,100,30);
calculate.addActionListener(this);
clear=new JButton("Clear");
clear.setBounds(200,200,100,30);
clear.addActionListener(this);
exit=new JButton("Exit");
exit.setBounds(340,200,100,30);
exit.addActionListener(this);
c.add(no1);
c.add(F1);
c.add(no2);
c.add(F2);
c.add(result);
c.add(F3);
c.add(calculate);
c.add(clear);
c.add(exit);
}
public void actionPerformed(ActionEvent ae)
{
 if(ae.getSource()==calculate)
 {

 if(F1.getText().equals("")||(F2.getText().equals("")) )
 {
 JOptionPane.showMessageDialog(fr,"Enter 2 integer numbers and click  calculate button");
 }
 else
 {
 int n1 = Integer.parseInt(F1.getText());
 int n2 = Integer.parseInt(F2.getText());
 int no4 = n1 + n2; // 10
 String s1 = String.valueOf(no4);
 F3.setText(s1);
 }
```

```
 }
if(ae.getSource()==clear)
{
F1.setText("");
F2.setText("");
F3.setText("");
}
if(ae.getSource()==exit)
{
 System.exit(0);
}
}
public static void main(String[] args) {
new addition();
}
}
```







Example program 2: Student Registration form

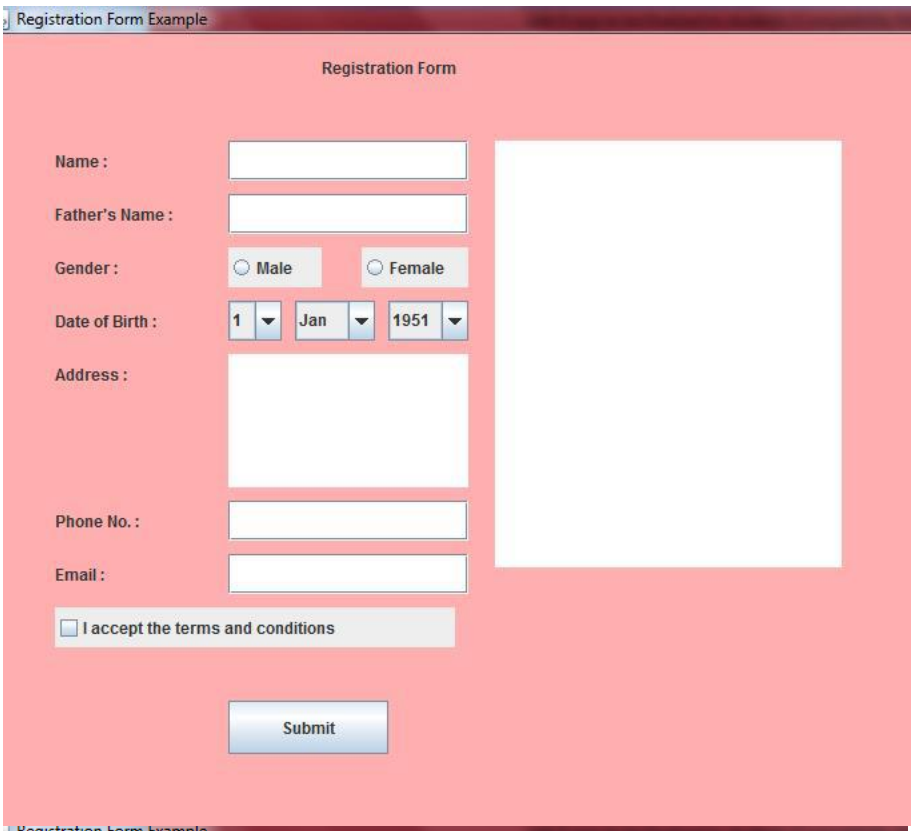
```
 import java.awt.*;
import java.awt.event.*;
```

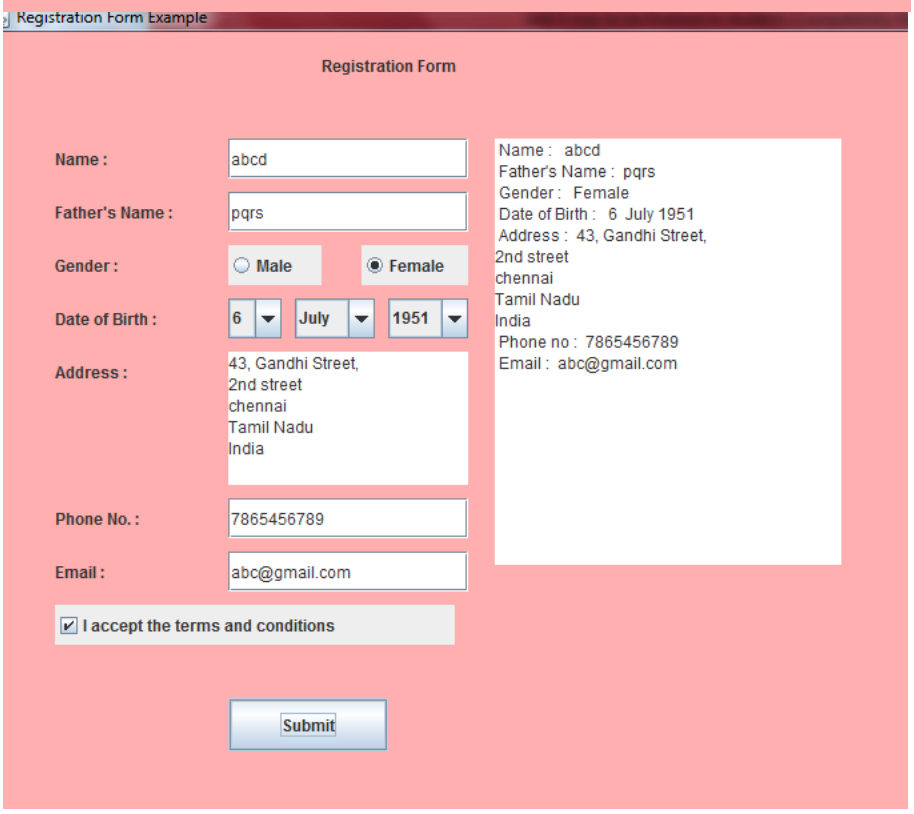
```
import javax.swing.*;
class testing1 extends JFrame implements ActionListener {
JTextField name_txt ;
 JTextField fname_txt;
JRadioButton male;
JRadioButton female;
 JComboBox day;
 JComboBox month;
 JComboBox year;
 JTextArea add_txtArea;
JTextField phone_txt;
JTextField email_txt;
JCheckBox chkbox;
 JButton submit_btn;
 JTextArea output_txtArea;
public testing1()
{
// Step 1 : Creating a frame using JFrame class
JFrame frame=new JFrame("Registration Form Example");  frame.setVisible(true);
frame.setSize(1000,1000);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// Step 2 : setting background color of Frame.
Container c=frame.getContentPane();
c.setLayout(null);
c.setBackground(Color.pink);

// step 3 : creating JLabel for Heading
JLabel heading_lbl=new JLabel("Registration Form"); heading_lbl.setBounds(250,5,200,40);

// Step 4 : Creating JLabel for Name
JLabel name_lbl=new JLabel("Name : ");
name_lbl.setBounds(50,80,100,30);
// Creating JTextField for Name
name_txt=new JTextField();
name_txt.setBounds(180,80,180,30);
// Step 5 : Creating JLabel for Father's Name
JLabel fname_lbl=new JLabel("Father's Name : ");
fname_lbl.setBounds(50,120,150,30);
// Creating JTextField for Father's name
fname_txt=new JTextField();
fname_txt.setBounds(180,120,180,30);
// Step 6 : Creating JLabel for Gender
JLabel gender_lbl=new JLabel("Gender : ");
gender_lbl.setBounds(50,160,100,30);

// Creating JRadioButton for the Male
male=new JRadioButton("Male");
male.setBounds(180,160,70,30);
 // Creating JRadioButton for the Female
female=new JRadioButton("Female");
female.setBounds(280,160,80,30);
 // Step 7 : Creating JLabel for Date of Birth
JLabel dob_lbl=new JLabel("Date of Birth : ");
dob_lbl.setBounds(50,200,100,30);
// Creating JComboBox for the day
String day_arr[]=new String[31];
for(int i=1;i<=31;i++)
```

```
day_arr[i-1]=Integer.toString(i);
day=new JComboBox(day_arr);
day.setBounds(180,200,40,30);
// Creating JComboBox for the month
String
month_arr[]={"Jan","Feb","March","April","May","June","July","Aug","Sept","Oct","No v","Dec" };
month=new JComboBox(month_arr);
month.setBounds(230,200,60,30);
// Creating JComboBox for the year
String year_arr[]=new String[70];
for(int i=1951;i<=2020;i++)
year_arr[i-1951]=Integer.toString(i);
year=new JComboBox(year_arr);
year.setBounds(300,200,60,30);
// Step 8 : Creating JLabel for the Address
JLabel add_lbl=new JLabel("Address : ");
add_lbl.setBounds(50,240,100,30);
// Creating JTextArea for the address
add_txtArea= new JTextArea();
add_txtArea.setBounds(180,240,180,100);
// Step 9 : Creating JLabel for the phone
JLabel phone_lbl=new JLabel("Phone No. : ");
phone_lbl.setBounds(50,350,100,30);
// Creating JTextField for the phone
phone_txt=new JTextField();
phone_txt.setBounds(180,350,180,30);
// Step 10 : Creating JLabel for the Email
JLabel email_lbl=new JLabel("Email : ");
email_lbl.setBounds(50,390,100,30);
// Creating JTextField for the Email
email_txt=new JTextField();
email_txt.setBounds(180,390,180,30);
// Step 11 : Creating JCheckBox for the license agreement  chkbox=new JCheckBox("I accept the terms and
conditions"); chkbox.setBounds(50,430,300,30);

// Step 12 : Creating JButton for submit the details submit_btn=new JButton("Submit");
submit_btn.setBounds(180,500,120,40);

// Step 13 : Adding ActionListener on submit button submit_btn.addActionListener(this);

// Step 14 : Creating JTextArea for output
output_txtArea=new JTextArea();
output_txtArea.setBounds(380,80,260,320);

// Step 15 : Adding label components to the container
c.add(heading_lbl);
c.add(name_lbl);
c.add(fname_lbl);
c.add(gender_lbl);
c.add(male);
c.add(female);
c.add(dob_lbl);
c.add(add_lbl);
c.add(phone_lbl);
c.add(email_lbl);
// Step 16 : Adding JTextField, JTextArea, JComboBox, JCheckBox, JRadioButton to the  container
c.add(name_txt);
```

```
c.add(fname_txt);
c.add(day);
c.add(month);
c.add(year);
c.add(add_txtArea);
c.add(phone_txt);
c.add(email_txt);
c.add(chkbox);
c.add(submit_btn);
c.add(output_txtArea);
} // ending of constructor
// step 17 :action to be performed when the button is clicked
public void actionPerformed(ActionEvent event)
{
 if(chkbox.isSelected()==true)
{
String name=name_txt.getText();
String fname=fname_txt.getText();
String gender="Male";
if(female.isSelected()==true)
 gender="Female";
String day_name=(String)day.getSelectedItem();
String month_name=(String)month.getSelectedItem();
String year_name=(String)year.getSelectedItem();
String add=add_txtArea.getText();
String phone=phone_txt.getText();
String email=email_txt.getText();

 // displaying value in the JTextArea
 output_txtArea.setText(" Name : " +name + "\n Father's Name : " +fname + "\n  Gender : "+gender + "\n
Date of Birth : "+day_name + " "+month_name + " "  +year_name + "\n Address : "+add + " \n Phone no :
"+phone + "\n Email : "+email  + "\n ");
 }
else
{
 output_txtArea.setText("Please accept the terms and condition and submit your  details");
}
}
public static void main(String args[])
{
 new testing1();
}
}
```

Sample Output:



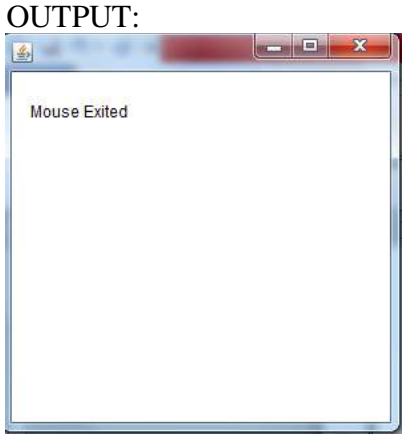


EXAMPLE 3:

```
import java.awt.*;
import java.awt.event.*;
public class MouseListenerExample extends Frame implements MouseListener  {
Label l;
MouseListenerExample()
{
addMouseListener(this);

l=new Label();
l.setBounds(20,50,100,20);
add(l);
setSize(300,300);
```

```
setLayout(null);
setVisible(true);
}
public void mouseClicked(MouseEvent e) {
l.setText("Mouse Clicked");
}
public void mouseEntered(MouseEvent e) {
l.setText("Mouse Entered");
}
public void mouseExited(MouseEvent e) {
l.setText("Mouse Exited");
}
public void mousePressed(MouseEvent e) {
l.setText("Mouse Pressed");
}
public void mouseReleased(MouseEvent e) {
l.setText("Mouse Released");
}
public static void main(String[] args) {
new MouseListenerExample();
}
}
```

OUTPUT:



Menus

Menus are a standard way for windowed desktop applications to allow users to select options. For example, applications typically have a File menu offering options (menu items) to save or open a file. In a windowed environment the user will use their mouse to navigate and select menu items. Menus and menu items typically also have the functionality of key combinations to select options, also known as keyboard shortcuts. In other words, key combinations allow quick menu selections without the need of a mouse.

Creating Menus and Menu Items

Before exploring how to invoke code triggered by selecting menu options, let's look at how to create menus. For starters, you must create a menu bar (javafx.scene.control.MenuBar) object to hold many javafx.scene.control.Menu objects. Each Menu object is similar to a tree hierarchal structure where Menu

objects can contain Menu and javafx.scene.control.MenuItem objects. Thus, a menu may contain other menus as submenus or menus within menus. MenuItems are child options within a Menu object. You can think of MenuItems as leaf nodes (containing no children) in a tree structure. Following list A shows the creation of a menu bar with a File menu that has a Save option as a menu item.

*List A: The creation of **MenuBar**, **Menu**, and **MenuItem** instances*

```
MenuBar menuBar = new MenuBar();
Menu fileMenu = new Menu("File");
fileMenu.getItems().add(new MenuItem("Save"));
menuBar.getMenus().add(fileMenu);
```

Figure below shows the output of above list, a simple File menu containing a Save menu item.

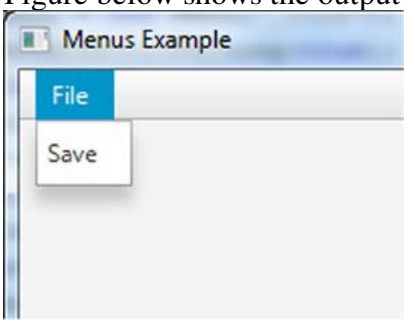


***Figure** A File menu containing a Save menu item*

Although you can create simple menu items, you may want more advanced options such as checked options or radio buttons. Based on the inheritance hierarchy, the following are subclasses of the MenuItem class. The following listing shows the available MenuItem subclasses to use as menu options.

A brief description of each subclass will follow.

- javafx.scene.control.CheckMenuItem
- javafx.scene.control.RadioMenuItem
- javafx.scene.control.CustomMenuItem
- javafx.scene.control.SeparatorMenuItem
- javafx.scene.control.Menu

A CheckMenuItem menu item is similar to a check box UI control, allowing the user to select items optionally. The RadioMenuItem menu item is similar to the radio button UI control, allowing the user to select only one item from an item group. When you want to create a custom menu item you can use the CustomMenuItem class. For instance, you may want to have a menu option that behaves like a toggle button. Next is a SeparatorMenuItem, which is really a derived class of type CustomMenuItem.

A SeparatorMenuItem is a menu item that displays as a visual line to separate menu items. Last in the list is the Menu class. Because a Menu class is a subclass of MenuItem, it has a getItems().add() method that's capable of adding children such as other Menu and MenuItem instances.

Invoking a Selected MenuItem

Now, that you know how to construct a menus and menu items, let's see how to invoke code that is attached to a menu item. You'll be happy to know that you wire-up handler code to a menu item in exactly the same way that you would wire-up JavaFX buttons, which means that menu items also have a setOnAction() method. The setOnAction() method receives a functional interface of type EventHandler<*ActionEvent*>, which is the handler code that is invoked when the menu item
is selected. Listing B shows two equivalent implementations of action code to be invoked when the Exit menu item is triggered. The first implementation uses an anonymous inner class, and the second uses Java 8's lambda expressions.

*Listing B. Adding handler code via the **setOnAction()** method*

```
// Implementation that uses an anonymous inner class
exitMenuItem.setOnAction(new EventHandler<ActionEvent>() {
@Override
public void handle(ActionEvent t) {
Platform.exit();
}
});
// Implementation that uses lambda expressions
exitMenuItem.setOnAction(ae -> Platform.exit());
```