Swing

Java Swing is a GUI Framework that contains a set of classes to provide more powerful and flexible GUI components than AWT. Java Swing is a part of Java Foundation Classes (JFC) that is used to create window-based applications. It is built on top of AWT API and acts as a replacement of AWT API, since it has almost every control corresponding to AWT controls. Swing provides the look and feel of modern Java GUI. Swing library is an official Java GUI tool kit released by Sun Microsystems. It is used to create graphical user interface with Java. Swing classes are defined in **javax.swing** package and its sub-packages.

MVC Architecture

Swing API architecture follows MVC architecture in the following manner.

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

Swing Features

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

The Swing classes are subclasses of **java.awt.Container** and **java.awt.Component.** The name of the Swing class starts with the letter **J** . the top-level class of Swing is **JComponent**. **JComponent** is both a container and a component. GUI components like button, label, checkbox, panel etc are handled in **JComponent** class. GUI components can be added on a panel window or a frame window. The frame in Swing is handled in **JFrame** class. The **JComponent** class and AWT class hierarchy is shown in figure 1.

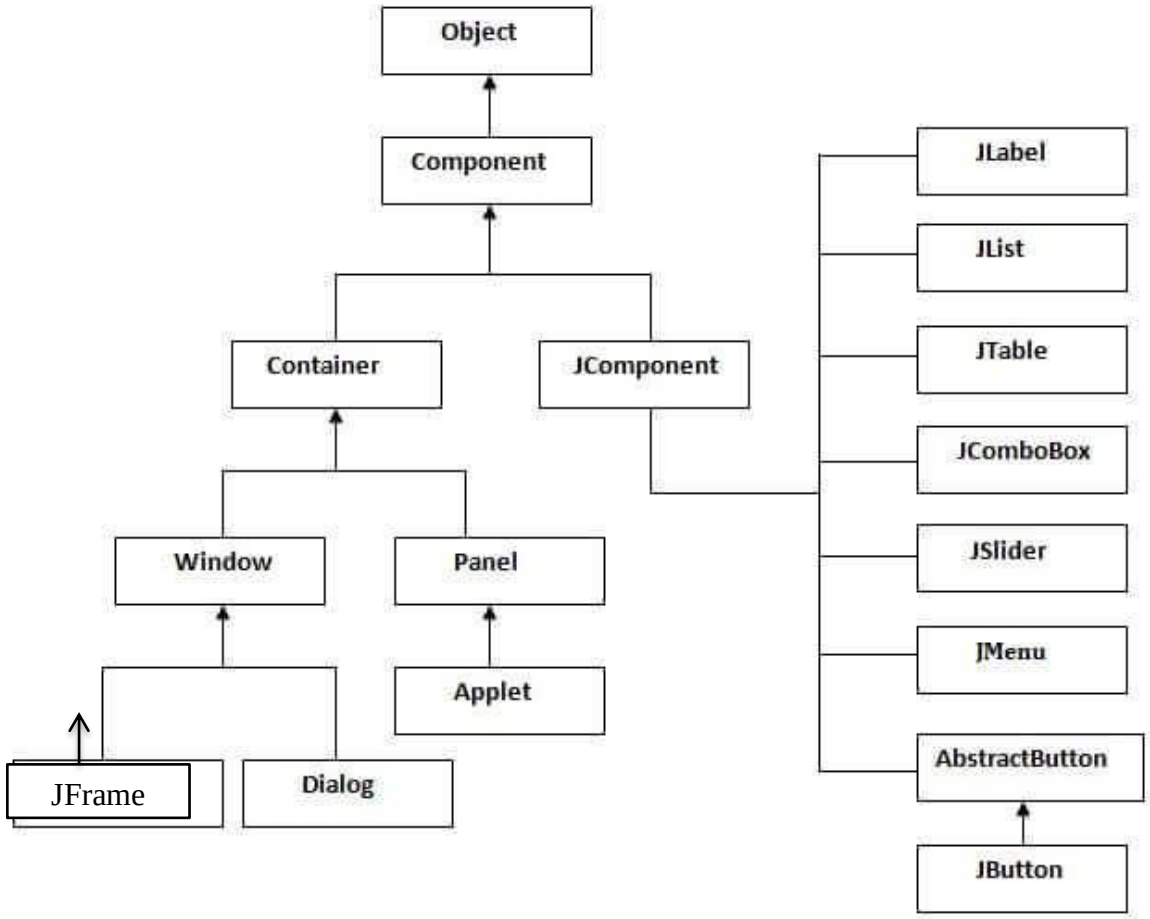


Figure 1: Swing Hierarchy

Difference between AWT and Swing

There are many differences between java awt and swing that are given below.

No.	Java AWT	Java Swing
1	Components are **platform-dependent**.	Java swing components are **platform-independent**.
2	Components are **heavyweight**.	Swing components are **lightweight**.
3	AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
4	AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT **doesn't follows MVC**(Model View Controller)	Swing **follows MVC**.

Swing components

A component is an independent visual control and Java Swing Framework contains a large set of these components which provide rich functionalities and allow high level of customization. They all are derived from **JComponent** class. All these components are lightweight components. This class provides some common functionality like pluggable look and feel, support for accessibility, drag and drop, layout, etc.

A container holds a group of components. It provides a space where a component can be managed and displayed. Containers are of two types:

1. Top level Containers
    - It inherits Component and Container of AWT.
    - It cannot be contained within other containers.
    - Heavyweight.
    - Example: **JFrame, JDialog, JApplet**
2. Lightweight Containers
    - It inherits JComponent class.
    - It is a general purpose container.
    - It can be used to organize related components together.
    - Example: **JPanel**

JFrame

JFrame is a java class that is extended by Frame class of Java. JFrame is treated as the main window. In JFrame different elements such as labels, text fields, buttons can be added. These elements on JFrame create Graphical User Interface. JFrame is also known as Swing top-level container.

Constructors of JFrame Class

Constructor	**Description**
JFrame()	**It constructs a new frame that is initially invisible.**
JFrame(GraphicsConfiguration gc)	It creates a Frame in the specified GraphicsConfiguration of a screen device and a blank title.
JFrame(String title)	**It creates a new, initially invisible Frame with the specified title.**

JFrame(String title, GraphicsConfiguration gc)	It creates a JFrame with the specified title and the specified GraphicsConfiguration of a screen device.

JButton

JButton class provides functionality of a button. It is used to create a labelled button component that has platform independent implementation. The application result in some action when the button is pushed. JButton class has three constuctors,

Constructor	**Description**
JButton()	It creates a button with no text and icon.
JButton(String s)	It creates a button with the specified text.
JButton(Icon i)	It creates a button with the specified icon object.

Commonly used methods of JButton class are

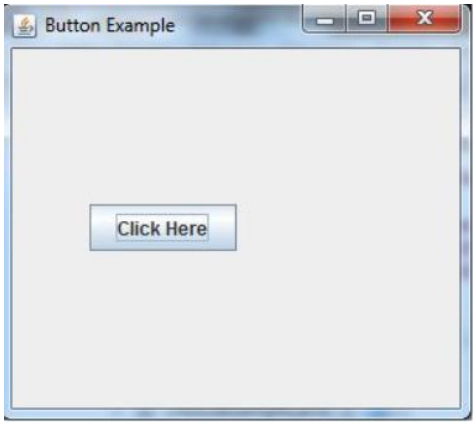
Method	**Description**
void setText(String s)	**It is used to set specified text on button**
String getText()	**It is used to return the text of the button.**
void setEnabled(boolean b)	**It is used to enable or disable the button.**
void setMnemonic(int a)	It is used to set the mnemonic on the button.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

Program 1: JButton Sample

```
import javax.swing.*;
public class ButtonExample
{
        public static void main(String[] args)
        {
                JFrame f=new JFrame("Button Example");
                JButton b=new JButton("Click Here");
```

```
                b.setBounds(50,100,95,30);
                f.add(b);
                f.setSize(400,400);
                f.setLayout(null);
                f.setVisible(true);
        }
}
```

Output



JLabel

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

Commonly used Constructors:

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

Commonly used Methods:

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

Program 2: JLabel Sample

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

Output

JCheckBox

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

Commonly used Constructors:

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

Commonly used Methods:

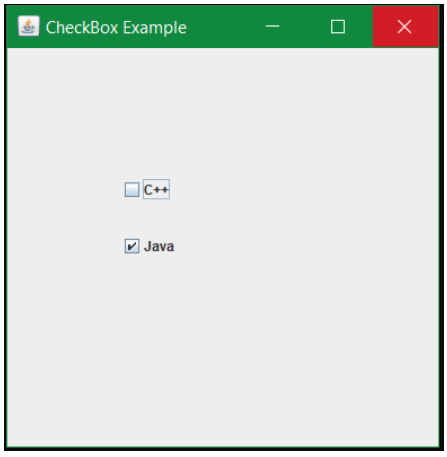
Methods	**Description**
String getText()	**It returns the label of the checkbox displays.**
void setText(String text)	**It sets the label of the checkbox.**
boolean isSelected()	**It returns the state of the checkbox.**
void setSelected(boolean state)	It sets the Checkbox to a specified state.

Program 3: JCheckBox Sample

```
import javax.swing.*;
public class CheckBoxExample
{
        CheckBoxExample()
```

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

Output



JRadioButton

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

Commonly used Constructors:

Constructor	Description

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

Commonly used Methods:

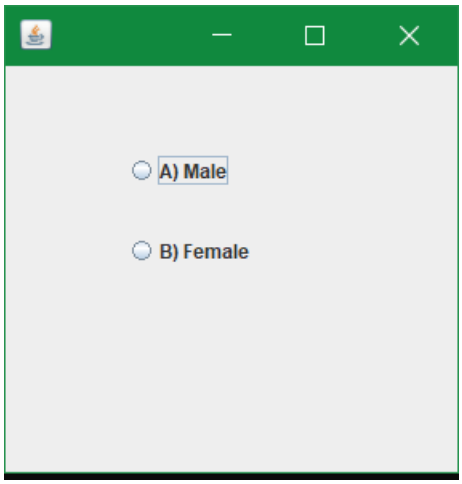
Methods	**Description**
void setText(String s)	It is used to set specified text on button.
String getText()	It is used to return the text of the button.
void setEnabled(boolean b)	It is used to enable or disable the button.
void addActionListener(ActionListener a)	It is used to add the action listener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

Program 4: JRadioButton Sample

```
import javax.swing.*;
public class RadioButtonEx
{
        JFrame f;
        JRadioButton r1,r2;
        ButtonGroup bg;
        RadioButtonEx()
        {
                f=new JFrame();
                r1=new JRadioButton("A) Male");
                r2=new JRadioButton("B) Female");
                r1.setBounds(75,50,100,30);
                r2.setBounds(75,100,100,30);
                bg=new ButtonGroup();
                bg.add(r1);bg.add(r2);
                f.add(r1);f.add(r2);
                f.setSize(300,300);
                f.setLayout(null);
                f.setVisible(true);
```

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

Output



Swing provides different classes to handle the text of different styles using the Swing model view concept. Basically, Swing's text component deals with two distinct types of text. One text component deals with simple text of one font and one color of text. The other type is a styled text with multiple fonts and multiple colors. The simple type texts are dealt by **JTextField, JPasswordField and JTextArea** classes. The styled texts are handled JEditorPane and JTextPane classes.

JTextField

JTextField is a subclass of JTextComponent, which is a subclass of JComponent

A JTextField object is a visual component that can display one line of editable text of one font and color at a time. The text is placed inside a box. The alignment of the text is defined by the following int type constants: JTextField.LEFT, JTextField.CENTER and JTextField.RIGHT.

The text field component is created using the following constructors:

Constructor	**Description**
JTextField()	Creates a new TextField
JTextField(String text)	Creates a new TextField initialized with the specified text.

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

Commonly used Methods:

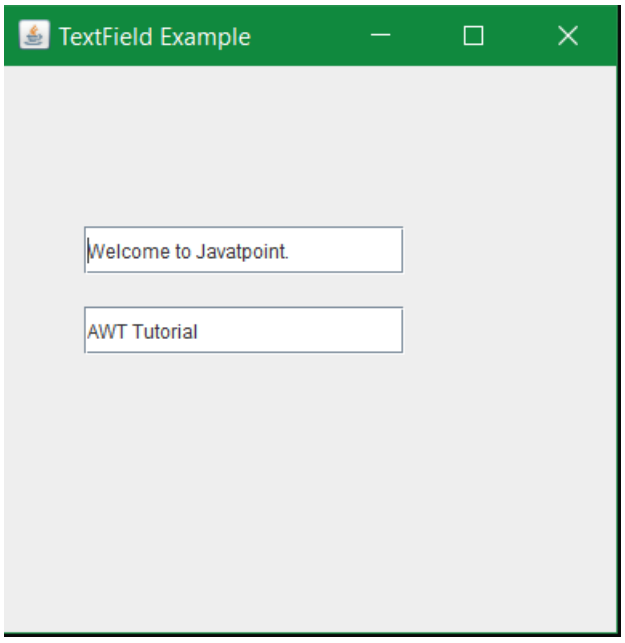
Methods	**Description**
void addActionListener (ActionListener l)	It is used to add the specified action listener to receive action events from this textfield.
void setText(String text)	It sets specified text as the text for this text filed.
void setFont(Font f)	It is used to set the current font.
String getText()	Returns the text contained in this textfield.
void setEditable(boolean edit)	Sets the textfield to editable (true) or not editable(false)

Program 5: JTextField Sample

```
import javax.swing.*;
class TextFieldExample
{
        JFrame f;
        JTextField t1,t2;

        TextFieldExample()
        {
                f= new JFrame("TextField Example");
                t1=new JTextField("Welcome to Javatpoint.");
                t1.setBounds(50,100, 200,30);
                t2=new JTextField("AWT Tutorial");
                t2.setBounds(50,150, 200,30);
                f.add(t1); f.add(t2);
                f.setSize(400,400);
                f.setLayout(null);
                f.setVisible(true);
```

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

Output



JPasswordField

The **JPasswordField** creates a display for text field similar to **JTextField**. The only difference is that when text is displayed, the actual characters are replaced by ***** character. This password field is useful where the text typed by a user is not to be seen by other people. JPasswordField is a subclass of **JTextComponent**. Like in JTextField, only one line a text with one font and color can be used.

The constructors used for creating password field are given below

Constructor	**Description**
JPasswordField()	**Creates an empty password field**
JPasswordField(int columns)	**Creates an empty password field with the specified number of columns**
JPassWordField(String text)	Create a password field with the specified text
JPasswordField(String text, int	Create a password field with the specified text and

columns)	specified number of columns

JPassword Field has a number of inherited methods and some of is own. Methods defined for JPasswordField are

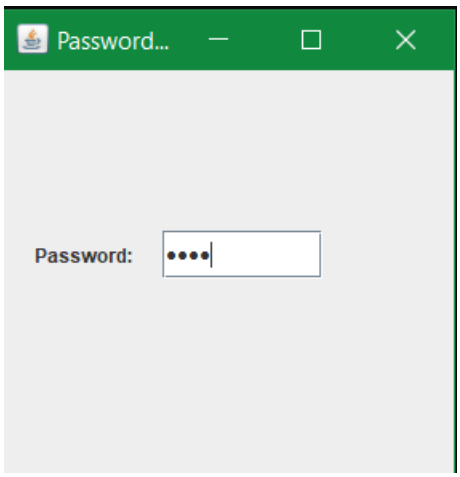
Methods	**Description**
boolean echoCharlsSet()	Returns true if an echo character has been set
char getEchoChar()	Returns the echo character set for this field.
void setEchoChar(char c)	Sets the specified character as echo character for this field

Program 6: JPasswordField Sample

```
import javax.swing.*;

public class PasswordFieldExample
{
        JFrame f;
        JPasswordField value;
        JLabel l1;
        PasswordFieldExample()
        {
                f=new JFrame("Password Field Example");
                value= new JPasswordField();
                l1=new JLabel("Password:");
                l1.setBounds(20,100, 80,30);
                value.setBounds(100,100,100,30);
                f.add(value);  f.add(l1);
                f.setSize(300,300);
                f.setLayout(null);
                f.setVisible(true);
        }
        public static void main(String[] args)
        {
                new PasswordFieldExample  ();
        }
```

```
}
```

Output



JText Area

JTextArea is a subclass of **JTextComponent**. A JTextArea component displays multiple lines of text in one color and with one font. Text area text is displayed as such in the defined window. There is no scroll bar to view the text. If the text is large, then a JScrollPane has to be created using the text area component.

JTextArea objects are created using the following constructors

Constructor	**Description**
JTextArea()	**Creates an empty text area**
JTextArea(int row, in column)	**Creates an empty text area with the specified number of rows and columns that are visible.**
JTextArea(String text, int row, int column)	Creates a text area with the specified text and with the specified rows and columns

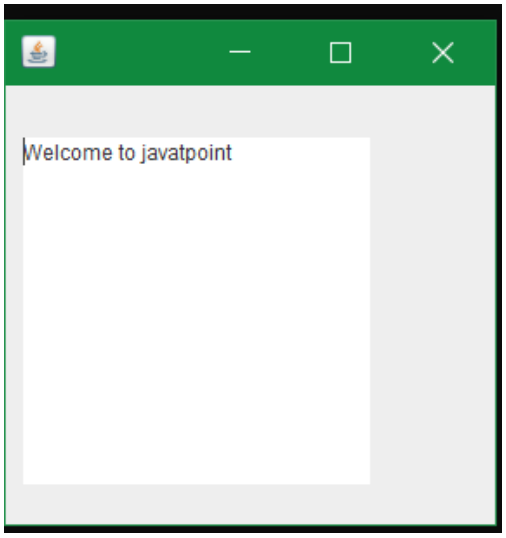
Commonly used methods are

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

Program 7: JTextArea Sample

```
import javax.swing.*;
public class TextAreaExample
{
        JFrame f;
        JTextArea area;
        TextAreaExample()
        {
                f= new JFrame();
                area=new JTextArea("Welcome to javatpoint");
                area.setBounds(10,30, 200,200);
                f.add(area);
                f.setSize(300,300);
                f.setLayout(null);
                f.setVisible(true);
        }
        public static void main(String args[])
        {
                new TextAreaExample();
        }
}
```

Output



JList

JList is part of Java Swing package . JList is a component that displays a set of Objects and allows the user to select one or more items . JList inherits JComponent class.

JList objects are created using the following constructors:

Constructor	**Description**
JList()	Creates a JList with an empty, read-only, model.
JList(ary[] listData)	Creates a JList that displays the elements in the specified array.

Commonly used methods are

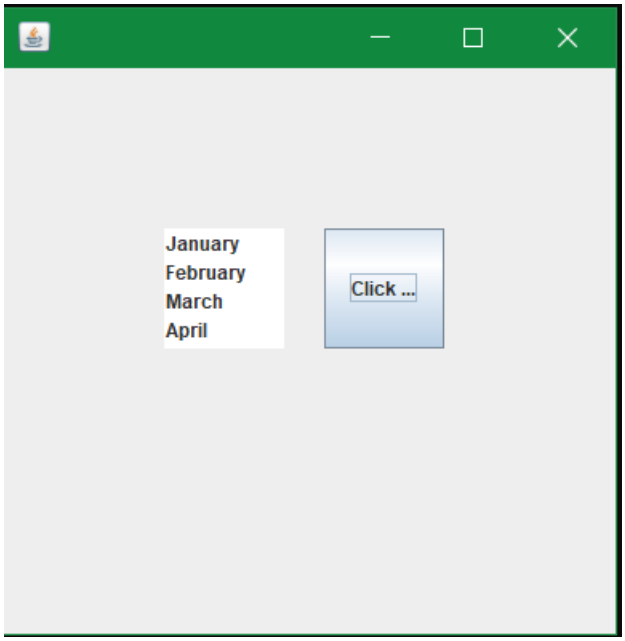
Methods	**Description**
getSelectedIndex()	**returns the index of selected item of the list**
getSelectedValue()	**returns the selected value of the element of the list**
setSelectedIndex(int i)	sets the selected index of the list to i
setSelectionBackground(Color c)	sets the background Color of the list
setSelectionForeground(Color c)	Changes the foreground color of the list
setVisibleRowCount(int v)	Changes the visibleRowCount property
setSelectedValue(Object a, boolean s)	selects the specified object from the list.

Program 8: JList with event handling example

```
import javax.swing.*;
import java.awt.event.*;
public class ListSample implements ActionListener
{
        JFrame f;
        JList l1;
        JButton b;
        ListSample()
        {
                f= new JFrame();
                String s[]={"January","February","March","April"};
                l1=new JList(s);
                b=new JButton("Click to select");
                l1.setBounds(100,100, 75,75);
```

```
                b.setBounds(200,100, 75,75);
                f.add(l1);
                f.add(b);
                b.addActionListener(this);
                f.setSize(400,400);
                f.setLayout(null);
                f.setVisible(true);
        }
        public void actionPerformed(ActionEvent ae)
        {
                System.out.println(l1.getSelectedIndex());
                System.out.println(l1.getSelectedValue());
        }
        public static void main(String args[])
        {
                new ListSample();
        }
}
```

Output



JComboBox

JComboBox is a part of Java Swing package. JComboBox inherits JComponent class. JComboBox shows a popup menu that shows a list and the user can select an option from that specified list. JComboBox can be editable or read- only depending on the choice of the programmer.

JComboBox objects are created using the following constructors:

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

Commonly used methods are

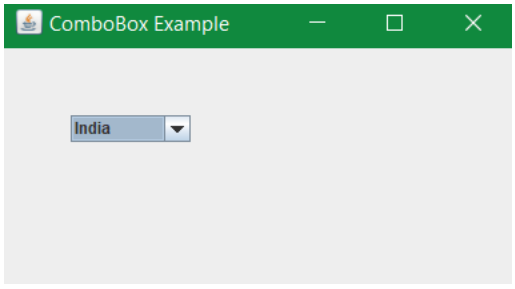
Methods	**Description**
void addItem(Object anObject)	**It is used to add an item to the item list.**
void removeItem(Object anObject)	**It is used to delete an item to the item list.**
void removeAllItems()	It is used to remove all the items from the list.
void setEditable(boolean b)	**It is used to determine whether the JComboBox is editable.**
void addActionListener(ActionListener a)	It is used to add the ActionListener.
void addItemListener(ItemListener i)	It is used to add the ItemListener.
getItemAt(int i)	returns the item at index i
getItemCount()	returns the number of items from the list
getSelectedItem()	returns the item which is selected
removeItemAt(int i)	removes the element at index i

Program 9: JComboBox example

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

```
public class ComboBoxEx
{
        JFrame f;
        ComboBoxEx()
        {
                f=new JFrame("ComboBox Example");
                String country[]=
                        {"India","Aus","U.S.A","England","Newzealand"};
                JComboBox cb=new JComboBox(country);
                cb.setBounds(50, 50,90,20);
                f.add(cb);
                f.setLayout(null);
                f.setSize(400,500);
                f.setVisible(true);
        }
        public static void main(String[] args)
        {
                new ComboBoxEx();
        }
}
```

Output



JPanel

The JPanel is a simplest container class. It provides space in which an application can attach any other component. It inherits the JComponents class. The main task of JPanel is to organize components, various layouts can be set in JPanel which provide better organisation of components, however it does not have a title bar.

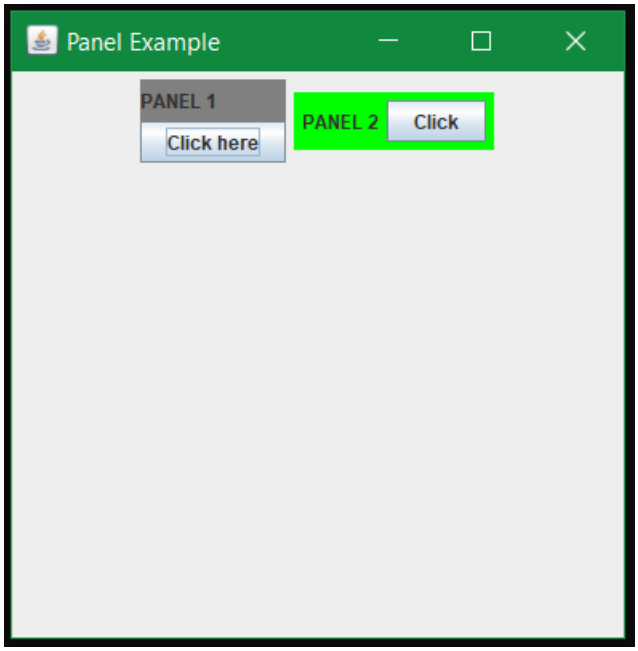
Constructor	**Description**
JPanel()	**It is used to create a new JPanel with a double buffer and a flow layout.**
JPanel(boolean isDoubleBuffered)	It is used to create a new JPanel with FlowLayout and the specified buffering strategy.
JPanel(LayoutManager layout)	**It is used to create a new JPanel with the specified layout manager.**

Program 10: JPanel example

```
import java.awt.*;
import javax.swing.*;
public class PanelSample
{
        public static void main(String args[])
    {
                JFrame f= new JFrame("Panel Example");
                JPanel p1=new JPanel(new GridLayout(2,1));
                JPanel p2=new JPanel();
                JLabel l1=new JLabel("PANEL 1");
                JButton b1=new JButton("Click here");
                JLabel l2=new JLabel("PANEL 2");
                JButton b2=new JButton("Click");

                p1.setBackground(Color.gray);
                p2.setBackground(Color.green);

                p1.add(l1);      p1.add(b1);
                p2.add(l2);      p2.add(b2);
                f.add(p1);f.add(p2);

                f.setLayout(new FlowLayout());
                f.setSize(400,400);
                f.setVisible(true);
```

```
        }
}
```

Output



Event Handling

EVENT: **-** In programming terms, an **event** is when something special happens. An event generally occurs when something changes within a graphical user interface. For forms, this means things like buttons being clicked, the mouse moving, text being entered into text fields, the programming closing, and a lot more. **Events** are objects in **Java**.

The graphical component generating the event is known as an **event source**. The event source will then send out an event object - it will contain information about the event.

The event object gets processed by an **event listener** assigned to the event source. Different types of graphical components have different types of **event listeners**.

Delegation Event model

The modern approach to handling events is based on the delegation event model, which defines standard and consistent mechanisms to generate and process events. Its concept is defined as follows: **a source generates an event and sends it to one or more listeners. In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns.**

The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is

able to "delegate" the processing of an event to a separate piece of code. In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them.

Events

In the delegation model, an event is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. 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. Many other user operations could also be cited as examples.

Events may also occur that are not directly caused by interactions with a user interface.

For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed. You are free to define events that are appropriate for your application.

Event Sources

A source is an object that generates an event. This occurs when the internal state of that object changes in some way. 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. Each type of event has its own registration method. Here is the general form:

```
public void addTypeListener(TypeListener el)
```

Here, Type is the name of the event, and el is a reference to the event listener. For example, the method that registers a keyboard event listener is called addKeyListener( ). The method that registers a mouse motion listener is called addMouseMotionListener( ). When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as multicasting the event. In all cases, notifications are sent only to listeners that register to receive them. Some sources may allow only one listener to register.

Event Listeners

A listener is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications. The methods that receive and process events are defined in a set of interfaces found in **java.awt.event**.

Event classes

The classes that represent events are at the core of Java's event handling mechanism. Main classes in **java.awt.event** are listed below.

Event class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract superclass for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component. MouseWheelEvent Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

The ActionEvent Class

An **ActionEvent** is generated when a button is pressed, a list item is double-clicked, or a menu item is selected. The **ActionEvent** class defines four integer constants that can be used to identify any modifiers associated with an action event: **ALT_MASK**, **CTRL_MASK**, **META_MASK**, and **SHIFT_MASK**. In addition, there is an integer constant, **ACTION_PERFORMED**, which can be used to identify action events.

ActionEvent has these three constructors:

ActionEvent(Object *src*, int *type*, String *cmd*)

ActionEvent(Object *src*, int *type*, String *cmd*, int *modifiers*)

ActionEvent(Object *src*, int *type*, String *cmd*, long *when*, int *modifiers*)

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*, and its command string is *cmd*. The argument *modifiers* indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. The *when* parameter specifies when the event occurred. You can obtain the command name for the invoking **ActionEvent** object by using the **getActionCommand( )** method, shown here:

String getActionCommand( )

For example, when a button is pressed, an action event is generated that has a command name equal to the label on that button.

The **getModifiers( )** method returns a value that indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. Its form is shown here:

int getModifiers( )

The method **getWhen( )** returns the time at which the event took place. This is called the event's *timestamp*. The **getWhen( )** method is shown here:

long getWhen( )

The FocusEvent Class

A **FocusEvent** is generated when a component gains or loses input focus. These events are identified by the integer constants **FOCUS_GAINED** and **FOCUS_LOST**. **FocusEvent** is a subclass of **ComponentEvent** and has these constructors:

FocusEvent(Component *src*, int *type*)

FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*)

FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*, Component *other*)

Here, *src* is a reference to the component that generated this event. The type of the event is specified by *type*. The argument *temporaryFlag* is set to **true** if the focus event is temporary. Otherwise, it is set to **false**. (A temporary focus event occurs as a result of another user interface operation. For example, assume that the focus is in a text field. If the user moves the mouse to adjust a scroll bar, the focus is temporarily lost.)

The other component involved in the focus change, called the *opposite component*, is passed in *other*. Therefore, if a **FOCUS_GAINED** event occurred, *other* will refer to the component that lost focus. Conversely, if a **FOCUS_LOST** event occurred, *other* will refer to the component that gains focus.

The ItemEvent Class

An **ItemEvent** is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected. There are two types of item events, which are identified by the following integer constants:

DESELECTED - The user deselected an item.

SELECTED- The user selected an item.

In addition, **ItemEvent** defines one integer constant, **ITEM_STATE_CHANGED**, that signifies a change of state.

ItemEvent has this constructor:

ItemEvent(ItemSelectable *src*, int *type*, Object *entry*, int *state*)

Here, *src* is a reference to the component that generated this event. For example, this might be a list or choice element. The type of the event is specified by *type*. The specific item that generated the item event is passed in *entry*. The current state of that item is in *state*

The KeyEvent Class

A **KeyEvent** is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants: **KEY_PRESSED**, **KEY_RELEASED**, and **KEY_TYPED**. The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated. Remember, not all keypresses result in characters. For example, pressing SHIFT does not generate a character.There are many other integer constants that are defined by **KeyEvent.** Here is one of its constructors:

KeyEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *code*, char *ch*)

Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the key was pressed is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when this key event occurred. The virtual key code is passed in *code*. The character equivalent is passed in *ch*. If no valid character exists, then *ch* contains **CHAR_UNDEFINED**. The **KeyEvent** class defines several methods, but the most commonly used ones are **getKeyChar( )**, which returns the

character that was entered, and **getKeyCode( )**, which returns the key code. Their general forms are shown here:

char getKeyChar( )

int getKeyCode( )

If no valid character is available, then **getKeyChar( )** returns **CHAR_UNDEFINED**. When a **KEY_TYPED** event occurs, **getKeyCode( )** returns **VK_UNDEFINED**.

The MouseEvent Class

There are eight types of mouse events. The **MouseEvent** class defines the following integer constants that can be used to identify them:

MOUSE_CLICKED -The user clicked the mouse.

MOUSE_DRAGGED -The user dragged the mouse.

MOUSE_ENTERED -The mouse entered a component.

MOUSE_EXITED -The mouse exited from a component.

MOUSE_MOVED -The mouse moved.

MOUSE_PRESSED- The mouse was pressed.

MOUSE_RELEASED- The mouse was released.

MOUSE_WHEEL -The mouse wheel was moved.

MouseEvent is a subclass of **InputEvent**. Here is one of its constructors:

MouseEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *x*, int *y*, int *clicks*, boolean *triggersPopup*)

Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when a mouse event occurred. The coordinates of the mouse are passed in *x* and *y*. The click count is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform.

Two commonly used methods in this class are **getX( )** and **getY( )**. These return the X and Y coordinates of the mouse within the component when the event occurred. Their forms are shown here:

int getX( )

int getY( )

Alternatively, you can use the **getPoint( )** method to obtain the coordinates of the mouse. It is shown here:

Point getPoint( )

It returns a **Point** object that contains the X,Y coordinates in its integer members: **x** and **y**.

The WindowEvent Class

There are ten types of window events. The **WindowEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

WINDOW_ACTIVATED	The window was activated.
WINDOW_CLOSED	The window has been closed.
WINDOW_CLOSING	The user requested that the window be closed.
WINDOW_DEACTIVATED	The window was deactivated.
WINDOW_DEICONIFIED	The window was deiconified.
WINDOW_GAINED_FOCUS	The window gained input focus.
WINDOW_ICONIFIED	The window was iconified.
WINDOW_LOST_FOCUS	The window lost input focus.
WINDOW_OPENED	The window was opened.
WINDOW_STATE_CHANGED	The state of the window changed.

WindowEvent is a subclass of **ComponentEvent**. It defines several constructors. The first is

WindowEvent(Window *src*, int *type*)

Here, *src* is a reference to the object that generated this event. The type of the event is *type*.

The next three constructors offer more detailed control:

WindowEvent(Window *src*, int *type*, Window *other*)

WindowEvent(Window *src*, int *type*, int *fromState*, int *toState*)

WindowEvent(Window *src*, int *type*, Window *other*, int *fromState*, int *toState*)

Sources of Events

Following Table lists some of the user interface components that can generate the events described

Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Event Listener Interfaces

As explained, the delegation event model has two parts: sources and listeners. Listeners are created by implementing one or more of the interfaces defined by the **java.awt.event** package. When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument.

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

The ActionListener Interface

This interface defines the **actionPerformed( )** method that is invoked when an action event occurs. Its general form is shown here:

void actionPerformed(ActionEvent *ae*)

The FocusListener Interface

This interface defines two methods. When a component obtains keyboard focus, **focusGained( )** is invoked. When a component loses keyboard focus, **focusLost( )** is called. Their general forms are shown here:

void focusGained(FocusEvent *fe*)

void focusLost(FocusEvent *fe*)

The ItemListener Interface

This interface defines the **itemStateChanged( )** method that is invoked when the state of an item changes. Its general form is shown here:

void itemStateChanged(ItemEvent *ie*)

The KeyListener Interface

This interface defines three methods. The **keyPressed( )** and **keyReleased( )** methods are invoked when a key is pressed and released, respectively. The **keyTyped( )** method is invoked when a character has been entered.

For example, if a user presses and releases the A key, three events are generated in sequence: key pressed, typed, and released. If a user presses and releases the HOME key, two key events

are generated in sequence: key pressed and released.

The general forms of these methods are shown here:

void keyPressed(KeyEvent *ke*)

void keyReleased(KeyEvent *ke*)

void keyTyped(KeyEvent *ke*)

The MouseListener Interface

This interface defines five methods. If the mouse is pressed and released at the same point, **mouseClicked( )** is invoked. When the mouse enters a component, the **mouseEntered( )** method is called. When it leaves, **mouseExited( )** is called. The **mousePressed( )** and **mouseReleased( )** methods are invoked when the mouse is pressed and released, respectively.

The general forms of these methods are shown here:

void mouseClicked(MouseEvent *me*)

void mouseEntered(MouseEvent *me*)

void mouseExited(MouseEvent *me*)

void mousePressed(MouseEvent *me*)

void mouseReleased(MouseEvent *me*)

The MouseMotionListener Interface

This interface defines two methods. The **mouseDragged( )** method is called multiple times as the mouse is dragged. The **mouseMoved( )** method is called multiple times as the mouse is moved. Their general forms are shown here:

void mouseDragged(MouseEvent *me*)

void mouseMoved(MouseEvent *me*)

The WindowListener Interface

This interface defines seven methods. The **windowActivated( )** and **windowDeactivated( )** methods are invoked when a window is activated or deactivated, respectively. If a window is iconified, the **windowIconified( )** method is called. When a window is deiconified, the **windowDeiconified()** method is called. When a window is opened or closed, the **windowOpened()** or **windowClosed()** methods are called, respectively. The **windowClosing( )** method is called when a window is being closed. The general forms of these methods are

void windowActivated(WindowEvent *we*)

void windowClosed(WindowEvent *we*)

void windowClosing(WindowEvent *we*)

void windowDeactivated(WindowEvent *we*)

void windowDeiconified(WindowEvent *we*)

void windowIconified(WindowEvent *we*)

void windowOpened(WindowEvent *we*)

LAYOUT MANAGERS

The Layout Managers are used to arrange components of frame in a particular manner. A layout manager arranges the child components of a container. It positions and sets the size of components within the container's display area according to a particular layout scheme. The layout manager's job is to fit the components into the available area, while maintaining the proper spatial relationships between the components.

AWT comes with a few standard layout managers that will collectively handle most situations.

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.

Each layout manager keeps track of a list of components that are stored by their names. The layout manager is notified each time you add a component to a container. Java has several predefined **LayoutManager** classes. You can use the layout manager that best fits your application.

FlowLayout

FlowLayout implements a simple layout style, which is similar to how words flow in a text editor. The direction of the layout is governed by the container's component orientation property, which, by default, is left to right, top to bottom. Therefore, by default, components are laid out line-by-line beginning at the center . In all cases, when a line is filled, layout advances to the next line. A small space is left between each component, above and below, as well as left and right. Here are the constructors for **FlowLayout**:

FlowLayout( )

FlowLayout(int *how*)

FlowLayout(int *how*, int *horz*, int *vert*)

The first form creates the default layout, which centers components and leaves five pixels of space between each component. The second form lets you specify how each line is aligned.Valid values for *how* are as follows:

FlowLayout.LEFT

FlowLayout.CENTER

FlowLayout.RIGHT

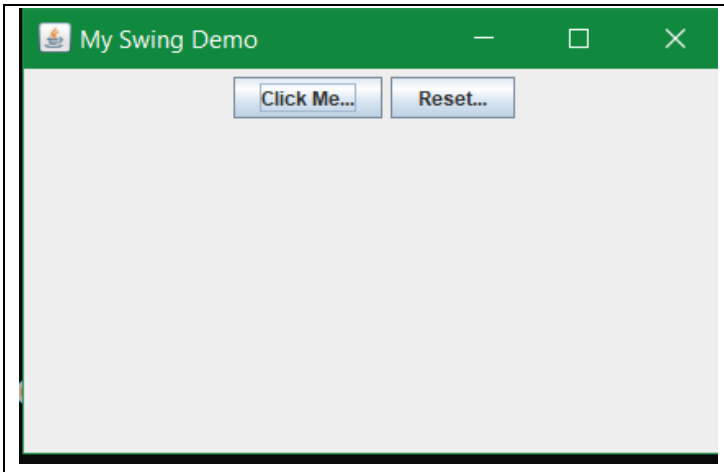
These values specify left, center, right, leading edge, and trailing edge alignment, respectively.

The third constructor allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively**.**

Example:FlowLayout

```
import javax.swing.*;
import java.awt.*;
class SwingDemo extends JFrame
{
        JButton b1,b2;
        SwingDemo()
        {
                super("My Swing Demo");
                setLayout(new FlowLayout());
                b1=new JButton("Click Me...");
                b2=new JButton("Reset...");
                add(b1);
                add(b2);
                setSize(500,600);
                setVisible(true);
        }
        public static void main(String args[])
        {
                new SwingDemo();
        }
}
```

Output



BorderLayout

The **BorderLayout** class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center. The four sides are referred to as north, south, east, and west. The middle area is called the center.

Here are the constructors defined by **BorderLayout**:

BorderLayout( )

BorderLayout(int *horz*, int *vert*)

The first form creates a default border layout. The second allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

BorderLayout defines the following constants that specify the regions:

BorderLayout.CENTER

BorderLayout.SOUTH

BorderLayout.EAST

BorderLayout.WEST

BorderLayout.NORTH

When adding components, you will use these constants with the following form of **add( )**, which is defined by **Container**:

void add(Component *compObj*, Object *region*)

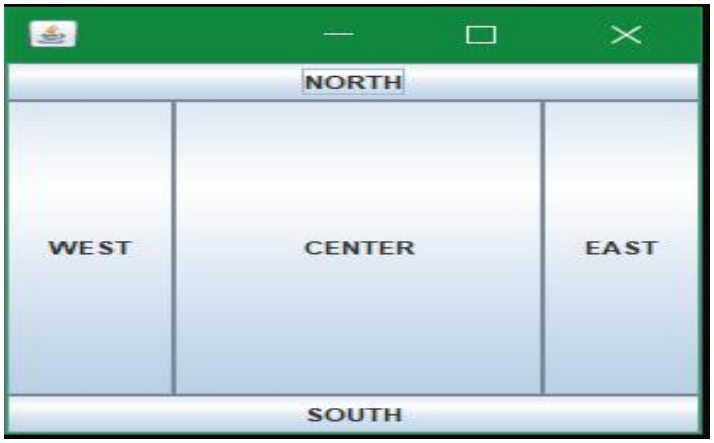
Here, *compObj* is the component to be added, and *region* specifies where the component will be added.

Example:BorderLayout

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

```
public class Border1
{
	JFrame f;
	Border1()
	{
		f=new JFrame();
		JButton b1=new JButton("NORTH");;
		JButton b2=new JButton("SOUTH");;
		JButton b3=new JButton("EAST");;
		JButton b4=new JButton("WEST");;
		JButton b5=new JButton("CENTER");;
		f.add(b1,BorderLayout.NORTH);
		f.add(b2,BorderLayout.SOUTH);
		f.add(b3,BorderLayout.EAST);
		f.add(b4,BorderLayout.WEST);
		f.add(b5,BorderLayout.CENTER);
		f.setSize(300,300);
		f.setVisible(true);
	}
	public static void main(String[] args)
	{
		new Border1();
	}
}
```

Output



GridLayout

GridLayout lays out components in a two-dimensional grid. When you instantiate a **GridLayout**, you define the number of rows and columns. The constructors supported by **GridLayout** are shown here:

GridLayout( )

GridLayout(int *numRows*, int *numColumns*)

GridLayout(int *numRows*, int *numColumns*, int *horz*, int *vert*)

The first form creates a single-column grid layout. The second form creates a grid layout with the specified number of rows and columns. The third form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. Either *numRows* or *numColumns* can be zero. Specifying *numRows* as zero allows for unlimited-length columns.

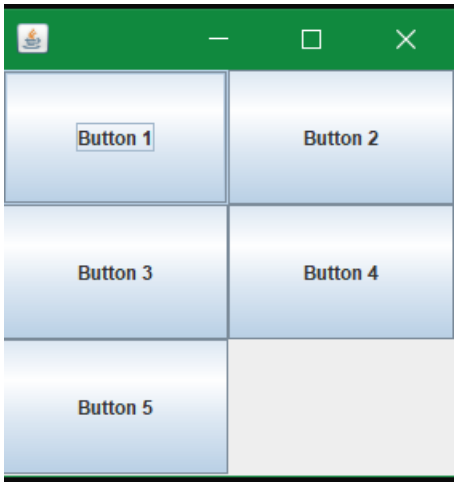
Specifying *numColumns* as zero allows for unlimited-length rows.

Example:GridLayout

```
import java.awt.*;
import javax.swing.*;
public class Grid
{
        JFrame f;
        JButton b1,b2,b3,b4,b5;
        Grid()
        {
                f=new JFrame();
                f.setLayout(new GridLayout(3,2));
                b1=new JButton("Button 1");
                b2=new JButton("Button 2");
                b3=new JButton("Button 3");
                b4=new JButton("Button 4");
                b5=new JButton("Button 5");
                f.add(b1);  f.add(b2);
                f.add(b3);  f.add(b4);
                f.add(b5);
                f.setSize(300,300);
```

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

Output



CardLayout

The **CardLayout** class is unique among the other layout managers in that 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. This can be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input. You can prepare the other layouts and have them hidden, ready to be activated when needed.

CardLayout provides these two constructors:

CardLayout( )

CardLayout(int *horz*, int *vert*)

The first form creates a default card layout. The second form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

Use of a card layout requires a bit more work than the other layouts. The cards are typically held in an object of type **Panel**. This panel must have **CardLayout** selected as its layout manager. The cards that form the deck are also typically objects of type **Panel**. Thus, you must create a panel that contains the deck and a panel for each card

in the deck. Next, you add to the appropriate panel the components that form each card. You then add these panels to the panel for which **CardLayout** is the layout manager. Finally, you add this panel to the window. Once these steps are complete, you must provide some way for the user to select between cards. One common approach is to include one push button for each card in the deck.

When card panels are added to a panel, they are usually given a name. Thus, most of the time, you will use this form of **add( )** when adding cards to a panel:

void add(Component *panelObj*, Object *name*)

Here, *name* is a string that specifies the name of the card whose panel is specified by *panelObj*.

After you have created a deck, your program activates a card by calling one of the following methods defined by **CardLayout**:

void first(Container *deck*)

void last(Container *deck*)

void next(Container *deck*)

void previous(Container *deck*)

void show(Container *deck*, String *cardName*)

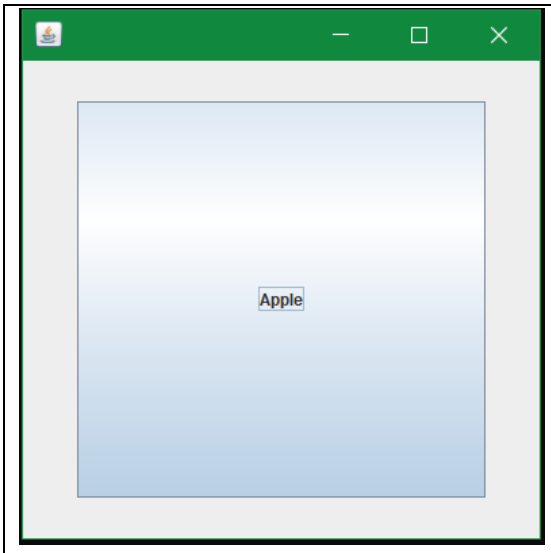
Here, *deck* is a reference to the container (usually a panel) that holds the cards, and *cardName* is the name of a card. Calling **first( )** causes the first card in the deck to be shown. To show the last card, call **last( )**. To show the next card, call **next( )**. To show the previous card, call **previous( )**. Both **next( )** and **previous()** automatically cycle back to the top or bottom of the deck, respectively. The **show( )** method displays the card whose name is passed in *cardName*.

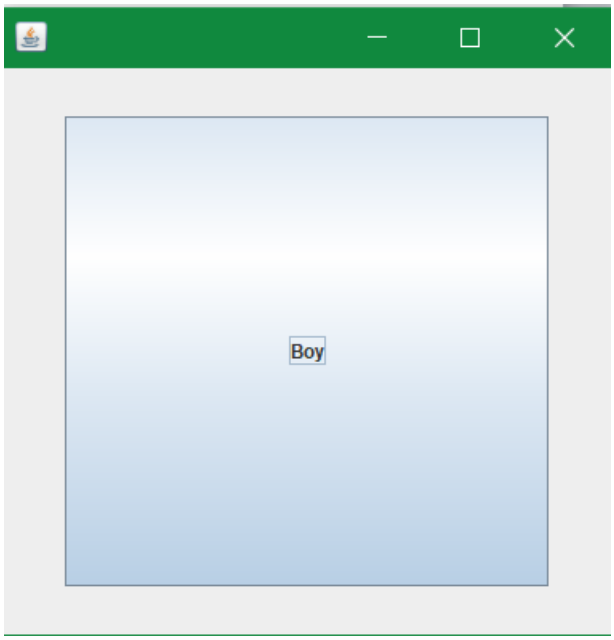
Example: CardLayout

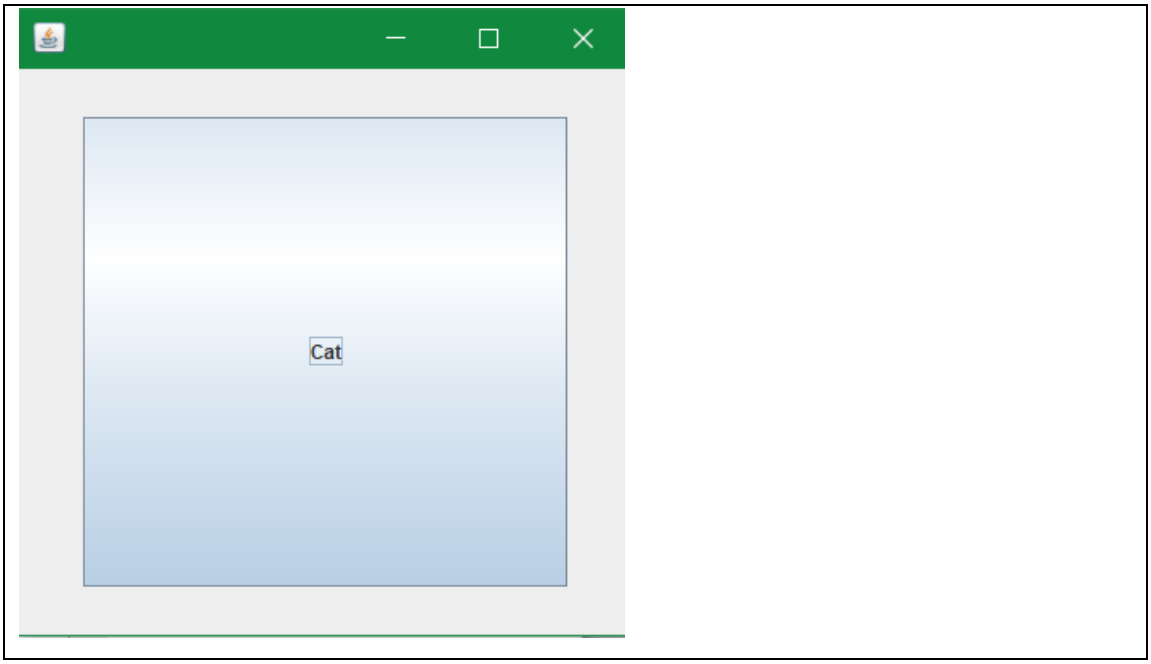
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class CardLayoutExample extends JFrame implements ActionListener
{
        CardLayout card;
        JButton b1,b2,b3;
        Container c;
```

```
        CardLayoutExample()
        {
                c=getContentPane();
                card=new CardLayout(40,30);
                //create CardLayout object with 40 hor space and 30 ver space
                c.setLayout(card);
                b1=new JButton("Apple");
                b2=new JButton("Boy");
                b3=new JButton("Cat");
                b1.addActionListener(this);
                b2.addActionListener(this);
                b3.addActionListener(this);
                c.add("a",b1);c.add("b",b2);c.add("c",b3);
        }
        public void actionPerformed(ActionEvent e)
        {
                card.next(c);
        }
        public static void main(String[] args)
        {
                CardLayoutExample cl=new CardLayoutExample();
                cl.setSize(400,400);
                cl.setVisible(true);
        }
}
```

Output


Apple


Boy



Box Layout

The **BoxLayout** class is used to arrange the components either vertically (along Y-axis) or horizontally (along X-axis). In BoxLayout class, the components are put either in a single row or a single column. The components will not wrap so, for example, a horizontal arrangement of components will stay horizontally arranged when the frame is resized. For this purpose, BoxLayout provides four constants. They are as follows:

public static final int X_AXIS

public static final int Y_AXIS

public static final int LINE_AXIS

public static final int PAGE_AXIS

Constructor of the **BoxLayout** class:

BoxLayout(Container c, int axis) **-** This creates a BoxLayout class that arranges the components with the X-axis or Y-axis.

Commonly Used Methods:

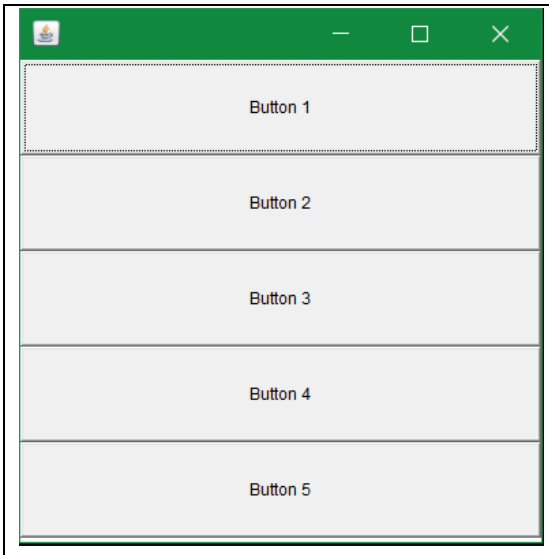
Method	**Description**
getLayoutAlignmentX (Container con)	Returns the alignment along the X axis for the container.
getLayoutAlignmentY (Container con)	Returns the alignment along the Y axis for the container

maximumLayoutSize (Container con)	Returns the maximum dimensions the target container can use to lay out the components it contains.
minimumLayoutSize (Container con)	Returns the minimum dimensions needed to lay out the components contained in the specified target container.
layoutContainer (Container tar)	Called by the AWT when the specified container needs to be laid out.

Example: BoxLayout – Y AXIS

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

public class BoxLayoutExample1 extends Frame
{
        Button buttons[];
        public BoxLayoutExample1 ()
        {
                buttons = new Button [5];
                for (int i = 0;i<5;i++)
                {
                        buttons[i] = new Button ("Button " + (i + 1));
                        add (buttons[i]);
                }
                setLayout (new BoxLayout (this, BoxLayout.Y_AXIS));
                setSize(400,400);
                setVisible(true);
        }
        public static void main(String args[])
        {
                BoxLayoutExample1 b=new BoxLayoutExample1();
        }
}
```

Output



Null Layout

The layout managers are used to automatically decide the position and size of the added components. In the absence of a layout manager, the position and size of the components have to be set manually. The **setBounds()** method is used in such a situation to set the position and size. To specify the position and size of the components manually, the layout manager of the frame can be null. Null layout is not a real layout manager. It means that no layout manager is assigned and the components can be put at specific x,y coordinates.

setBounds()

The setBounds() method needs four arguments. The first two arguments are x and y coordinates of the top-left corner of the component, the third argument is the width of the component and the fourth argument is the height of the component.

Syntax

```
setBounds(int x-coordinate, int y-coordinate, int width, int height)
```

Example : Null Layout

```
import javax.swing.*;
import java.awt.*;
public class SetBoundsTest
{
        public static void main(String arg[])
        {
```

```
                JFrame frame = new JFrame("SetBounds Method Test");
                frame.setSize(375, 250);
                frame.setLayout(null);        // Setting layout as null
                JButton button = new JButton("Hello Java");

                button.setBounds(80,30,120,40);      // Setting position and size of a
button
                frame.add(button);
                frame.setVisible(true);
   }
}
```

Output