

Unit-6

Syllabus:

AWT: introduction, components and containers, Button, Label, Checkbox, Radio Buttons, List Boxes, Choice Boxes, Container class, Layouts, Menu and Scrollbar.

Swing: Introduction, JFrame, JApplet, JPanel, Components in Swings, Layout Managers in Swings, JList and JScrollPane, Split Pane, JTabbedPane, JTree, JTable, Dialog Box.

Objectives:

- ✓ Discuss AWT Components, containers, Layouts, Menu and scrollbar
- ✓ Explain Swings JFrame, JApplet, JPanel, Components.
- ✓ Illustrate Layout Managers
- ✓ Explain JList, JScrollPane, SplitPane, JTabbedPane, JTree, JTable, DialogBox.

Outcomes:

- Demonstrate AWT Components
- Examine Swings Components,
- Distinguish JScrollPane, SplitPane, JTabbedPane.
- Classify JTree, JTable, DialogBox

Unit-6 previous questions

Small Questions:

- 1) Differentiate GridLayout and GrodBagLayout. [Set-1April – 2018-R16].
- 2) What are the types of check boxes present in awt. [Set-2, April–18-R16].
- 3) Write different types of controls supported by awt. [Set-3, April–18-R16].
- 4) What are the subclasses of Container class? [Set-4, April–18-R16].
- 5) Give a note on layouts in AWT. [Set-1, April –18-R13].
- 6) Write a java program that makes a window with a scroll bar at the right side of the window. [Set-2, April – 18-R13].
- 7) Why layouts are needed? [Set-3, April –18-R13].
- 8) List the controls supported by AWT. [Set-4, April –18-R13].

Big Questions:

- 1) Explain different types Layout managers present in AWT with sample programs. [Set-1April – 2018-R16].
- 2) How do you change the current layout manager for a container? [Set-4April – 2018-R16].
- 3) Write a program in awt to design the registration form. [Set-4April – 2018-R16].
- 4) Write a program to design calculator using awt. [Set-2April – 2018-R16].
- 5) Explain various event adopter classes in awt and also give their syntaxes in java. [Set-2April – 2018-R16].
- 6) Differentiate the following [Set-3, April –18-R13].
 - i) TextField and TextArea.
 - ii) Menu and MenuItem.
 - iii) Checkbox and CheckboxGroup

Introduction to AWT (Abstract Window Toolkit)

AWT Classes

The AWT classes are contained in the java.awt package. It is one of Java's largest packages. Fortunately, because it is logically organized in a top-down, hierarchical fashion, it is easier to understand and use than you might at first believe lists some of the many AWT classes.

- The AWT contains numerous classes and methods that allow us to create and manage windows.
- Main purpose of AWT:
 - To support applet windows.

AWT Classes : java.awt package

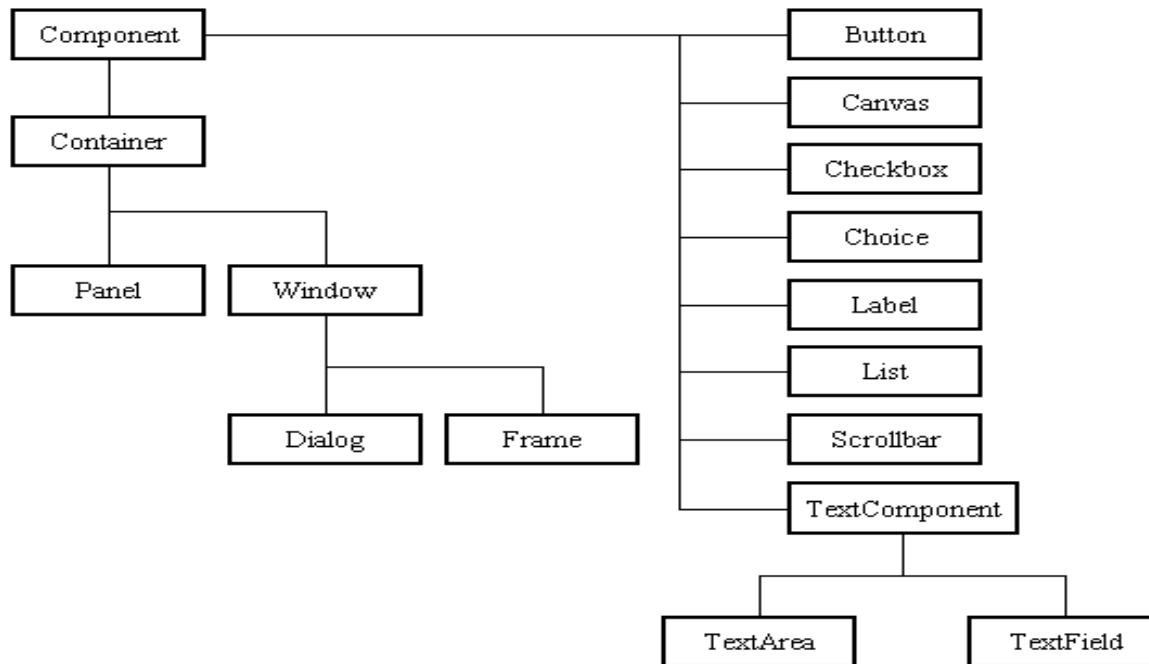
Class	Description
1. AWTEvent	Encapsulates AWT events.
2. BorderLayout	The border layout manager. Border layouts use five components: North, South, East, West, and Center.
3. Button	Creates a push button control.
4. Canvas	A blank, semantics-free window.
5. CardLayout	The card layout manager. Card layouts emulate index cards. Only the one on top is showing.
6. Checkbox	Creates a check box control.
7. CheckboxGroup	Creates a group of check box controls.
8. Choice	Creates a pop-up list.
9. Color	Manages colors in a portable, platform- independent fashion.
10. Component	An abstract superclass for various AWT components.
11. Container	A subclass of Component that can hold other components.
12. Dialog	Creates a dialog window.
13. Event	Encapsulates events.
14. FlowLayout	The flow layout manager. Flow layout positions components left to right, top to bottom.
15. Font	Encapsulates a type font
16. Frame	Creates a standard window that has a title bar, resize corners, and a menu bar.
17. Graphics	Encapsulates the graphics context. This context is used by the various output methods to display output in a window.
18. GridBagLayout	The grid bag layout manager. Grid bag layout displays components subject to the constraints specified by GridBagConstraints .
19. GridLayout	The grid layout manager. Grid layout displays components in a two- dimensional grid.
20. Label	Creates a label that displays a string.
21. List	Creates a list from which the user can choose. Similar to the standard Windows list box.
22. Menu	Creates a pull-down menu.
23. Scrollbar	Creates a scroll bar control.
24. TextArea	Creates a multiline edit control.
25. TextComponent	A superclass for TextArea and TextField .
26. TextField	Creates a single-line edit control.
27. Toolkit	Abstract class implemented by the AWT.

Window Fundamentals:

- The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level.
- The 2 most common windows are:
 1. Those derived from Panel, which is used by applets.
 2. Those derived from Frame, which creates a standard window.

class hierarchy for Panel and Frame:

class hierarchy for Panel and Frame :



Component :

- Component class is at the top most class of the AWT hierarchy.
- A component is an object having graphical representation that can be displayed on the screen and interact with the user. Examples of components are buttons ,checkboxes etc.
- All user interface elements that are displayed on the screen and that interact with the user are subclasses of Component.
- Any thing that is derived from the component class is can be a component.
- The class contains no of methods for event handling,manging graphics painting and sizing and resizing the window.

Container :

- Container class is a subclass of Component.
- It has additional methods that allow other Component objects to be nested within it.
- A container object is component that can contain other AWT components.
- Some of the containers are Frames,Dialogs,windows,panel(applet)
- Responsibility of a container : To lay - out (that is, positioning) any components that it contains.

Panel :

- The Panel class is a concrete subclass of Container.
- It doesn't add any new methods; it simply implements Container.
- A panel provides space in which an application can attach any other component, including other panels
- It is a concrete screen component.

- Applet class is the sub class of panel
- Applet's output is drawn on the surface of a Panel object.
- In essence, a Panel is a window that does not contain a title bar, menu bar, or border.
- This is why we don't see these items when an applet is run inside a browser.
- When we run an applet using an appletviewer, the applet viewer provides the title and border.
- Other components can be added to a Panel object by its add() method (inherited from Container).
- Once these components have been added, we can position and resize them manually using the setLocation(), setSize(), or setBounds() methods defined by Component.

Window :

- The Window class creates a top-level window.
- A window objects is a top-level window with no borders and no menu bar.
- It doesn't contain any components.
- It has two subclasses; Frame is window with name and menu bar and Dialog to represent dialog boxes.

Frame :

- It is a subclass of Window and has a title bar, menu bar, borders, and resizing corners. 2 forms of Frame constructors :

1. Frame() : It creates a standard window that does not contain a title. Frame f=new Frame();

1. Frame(String *title*) : It creates a window with the title specified by *title*.
Frame f=new Frame("my frame");

Note : we cannot specify the dimensions of the window. Instead, we must set the size of the window after it has been created.

- Another way is to create a subclass myframe to the Frame class and create an object to the sub class
- Ex
- Class Myframe extends frame
- {
- Myframe m=new Myframe();
- }

Methods in Frames :

Setting the Window's Dimensions :

- setSize() : used to set the dimensions of the window.
Syntax : void setSize(int *newWidth*, int *newHeight*) void setSize(Dimension *newSize*)

where, new size of window is specified by *newWidth* and *newHeight* or by the width and height fields of the Dimension object passed in *newSize*. (dimensions → pixels).

- getSize() : used to obtain the current size of a window.
Syntax : Dimension getSize()

Note : It returns the current size of the window contained within the width and height fields of a Dimension object.

Hiding & Showing a Window :

- setVisible() : After a frame window has been created, it will not be visible until we call setVisible()
Syntax : void setVisible(boolean *visibleFlag*)
Note : The component is visible if the argument to this method is true. Otherwise, it is hidden.

Setting a Window's Title :

- setTitle() : We can change the title in a frame window using setTitle()
Syntax : void setTitle(String *newTitle*)
Here, *newTitle* is the new title for the window.

Labels

- The easiest control to use is a label.
- A *label* is an object of type **Label**, and it contains a string, which it displays.

- Labels are passive controls that do not support any interaction with the user.

-

Label defines the following constructors:

```
Label( )
Label(String
str)
Label(String str, int how)
```

- The first version creates a blank label.
- The second version creates a label that contains the string specified by *str*. This string is left-justified.
- The third version creates a label that contains the string specified by *str* using the alignment specified by *how*. The value of *how* must be one of these three constants: **Label.LEFT**, **Label.RIGHT**, or **Label.CENTER**.

setText() :It is used to set or change the text in a label .

getText():It is used to obtain the current label. These methods are shown here:

```
void setText(String
str) String getText( )
```

For **setText()**, *str* specifies the new label. For **getText()**, the current label is returned. **setAlignment()** :It is used to set the alignment of the string within the label . **getAlignment()**:It is used To obtain the current alignment,

```
void setAlignment(int
how) int getAlignment( )
```

Here, *how* must be one of the alignment constants shown earlier.

The following example creates three labels and adds them to an applet window:

```
// Demonstrate
Labels import
java.awt.*; import
java.applet.*;
/*
<applet code="LabelDemo" width=300 height=200>
</applet>
*/
public class LabelDemo extends
Applet { public void init() {
Label one = new Label("One");
Label two = new Label("Two");
Label three = new
Label("Three");
// add labels to applet
window add(one);
add(two);
add(three)
;
}
}
```

Here is the window created by the **LabelDemo** applet. Notice that the labels are organized in the window by the default layout manager. Later, you will see how to control more precisely the placement of the labels.



Buttons:

- Perhaps the most widely used control is the push button.
- A *push button* is a component that contains a label and that generates an event when it is pressed.
- Push buttons are objects of type **Button**.

Button defines these two constructors:

Button()
Button(String
str)

- The first version creates an empty button.
- The second creates a button that contains *str* as a label.

setLabel():

It is used to set label to button

getLabel():

It is used to get the label on the button

```
void setLabel(String  
str) String getLabel()
```

- Here, *str* becomes the new label for the button.

Handling Buttons

- Each time a button is pressed, an action event is generated.
- This is sent to any **ActionListener** that previously registered to receive action event notifications from that component.
- **ActionListener** interface defines the **actionPerformed()** method, which is called when an event occurs.
- An **ActionEvent** object is supplied as the argument to this method
- It contains both a reference to the button that generated the event and label of the button.

Program: write a java applet that reads two numbers from two separate textfields and display sum of two numbers in third textfield when button "add" is pressed?

Programcode:

```
import java.awt.*;  
import java.applet.*;  
import  
java.awt.event.*;  
/*  
<applet code="add" width=500 height=500>  
</applet>
```

*/

```
public class add extends Applet implements ActionListener
```

```
{
```

```
    TextField t1,t2,t3;
```

```
    Label l1,l2,l3;
```

```
    Button b;
```

```
    public void init()
```

```
    {
```

```
        l1=new Label("Enter a
```

```
        value"); t1=new
```

```
        TextField(12);
```

```
        add(l1);          add(t1);
```

```
        l2=new Label("enter b
```

```
        value"); t2=new
```

```
        TextField(12);
```

```
        add(l2);          add(t2);
```

```
        l3=new Label("Result");
```

```
        t3=new TextField(12);
```

```
        add(l3);          add(t3);
```

```
        b=new Button("Add");
```

```
        add(b);
```

```
        b.addActionListener(this);
```

```
    }
```

```
    public void actionPerformed(ActionEvent ae)
```

```
    {
```

```
        String n1 = t1.getText();
```

```
        String n2 =
```

```
        t2.getText(); int a=
```

```
        Integer.parseInt(n1);
```

```
        int b=
```

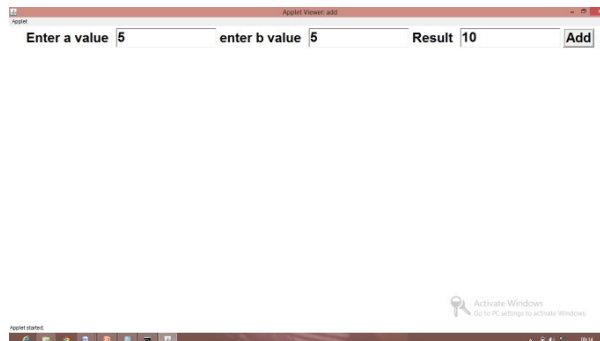
```
        Integer.parseInt(n2); int
```

```
        sum=a+b;
```

```
        t3.setText(Integer.toString(sum));
```

```
    }
```

```
}
```



**Check
Boxes**

- *Acheck box* is a control that is used to turn an option on or off.
- It consists of a small box that can either contain a check mark or not.
- There is a label associated with each check box that describes what option the box represents. You change the state of a check box by clicking on it.

- Check boxes can be used individually or as part of a group.
- Check boxes are objects of the **Checkbox** class.

Checkbox supports these constructors:

```
Checkbox( )
Checkbox(String
str)
Checkbox(String str, boolean on)
Checkbox(String str, boolean on, CheckboxGroup
cbGroup)
Checkbox(String str, CheckboxGroup
cbGroup, boolean on)
```

- The first form creates a check box whose label is initially blank. The state of the check box is unchecked.
- The second form creates a check box whose label is specified by *str*. The state of the check box is unchecked.
- The third form allows you to set the initial state of the check box. If *on* is **true**, the check box is initially checked; otherwise, it is cleared.
- The fourth and fifth forms create a check box whose label is specified by *str* and whose group is specified by *cbGroup*.

- *Methods:*

getState() : To retrieve the current state of a check box.

setState() : To set its state.

getLabel() : we can obtain current label associated with a check box

setLabel() : To set the label

Syntax :

```
boolean getState( )
void setState(boolean
on)
String getLabel( )
void setLabel(String str)
```

here,

on → **true**, the box is checked. on → **false**, the box is cleared.

str → The string passed in *str* becomes the new label associated with the
invoking check box.

box.

Handling Check Boxes:

- Each time a check box is selected or deselected, an item event is generated.
- This is sent to **ItemListener** that previously registered to receive an item event notifications from that component.
- **ItemListener** interface defines the **itemStateChanged()** method.
- An **ItemEvent** object is supplied as the argument to this method. It contains information about the event

Example program:

```
import java.awt.*;
import java.applet.*;
import
java.awt.event.*;
/*
<applet code="check" width=1000 height=1000>
```

```
</applet>  
*/
```

```
public class check extends Applet implements  
    ItemListener { String msg,msg1,msg2,msg3;
```

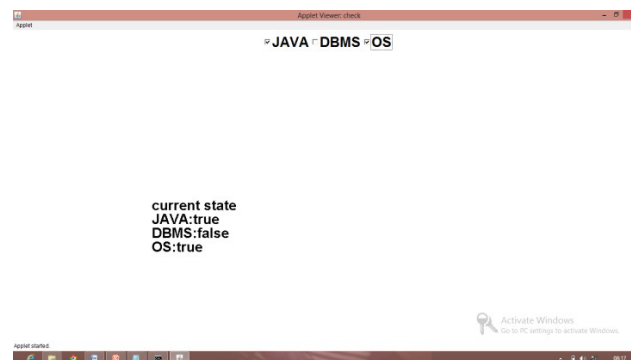
```

        Checkbox c1,c2,c3;
    public void init()
    {
        c1=new Checkbox("JAVA");
        c2=new
        Checkbox("DBMS");
        c3=new Checkbox("OS");
        add(c1);
        add(c2);
        add(c3);
        c1.addItemListener(thi
s);
        c2.addItemListener(thi
s);
        c3.addItemListener(thi
s);
    }
    public void itemStateChanged(ItemEvent ie)
    {repaint();
    }

    public void paint(Graphics g)
    { msg="current state";
      g.drawString(msg,300,390);
      msg1="JAVA:"+c1.getState()
      ;
      g.drawString(msg1,300,420);
      msg2="DBMS:"+c2.getState
      ();
      g.drawString(msg2,300,450);
      msg3="OS:"+c3.getState();
      g.drawString(msg3,300,480);
    }
}

```

Output:



CheckboxGroup : Radio buttons

- It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called *radio buttons*.
- To create a set of mutually exclusive check boxes, we must first define the group to which they will belong and then specify that group when we construct the check boxes.
- Check box groups are objects of type `CheckboxGroup`. `CheckboxGroup cbg=new CheckboxGroup();`

`getSelectedCheckbox()` : It determines which check box in a group is currently selected.

setSelectedCheckbox() : It sets a check box.

Syntax :

Checkbox setSelectedCheckbox()

void setSelectedCheckbox(Checkbox

which) here,

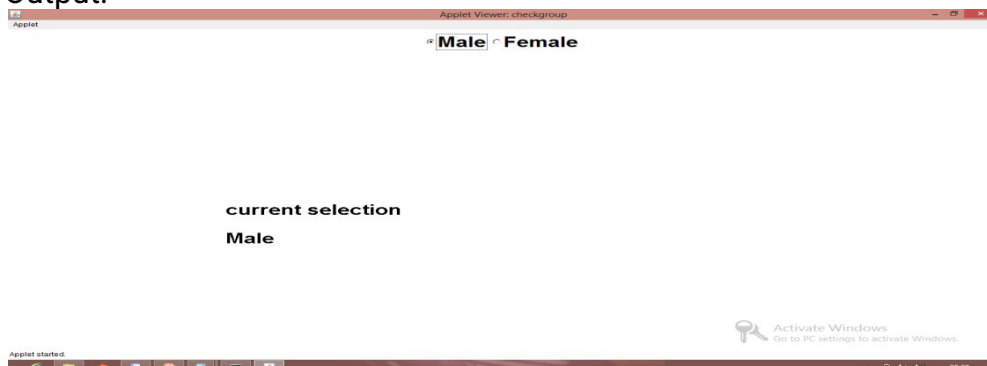
which → check box that we want to be selected.

Example Program:

```
import java.awt.*;
import java.applet.*;
import
java.awt.event.*;
/*
<applet code="checkgroup" width=1000 height=1000>
</applet>
*/
public class checkgroup extends Applet implements ItemListener {

    String
        msg,msg1,msg2;
        Checkbox m,f;
        CheckboxGroup
cbg; public void init()    {
    cbg=new CheckboxGroup();
    m=new
    Checkbox("Male",cbg,true);
    f=new
    Checkbox("Female",cbg,false);
    add(m);
    add(f);
    m.addItemListener(this);
    f.addItemListener(this);
}
    public void itemStateChanged(ItemEvent ie)
        { repaint();
        }
}
public void paint(Graphics g)
{ msg="current selection";
g.drawString(msg,300,390);
msg2=cbg.getSelectedCheckbox().getLabel();
g.drawString(msg2,300,450);
}
}
```

Output:



Choice Controls

- The **Choice** class is used to create a *pop-up list* of items from which the user may choose.
- **Choice** control is a form of menu
- Choice defines only default constructor, which creates an empty list. Syntax : Choice()

add() : Used to add item to the choice

list. Syntax : void add(String *name*)

here, *name* → name of item being added.

Note : Items are added to the list in the order in which calls to add() occur.

Choice c=new

Choice(); c.add("CSE");

getSelectedItem() or getSelectedIndex() : To determine which item is currently selected

Syntax :

String getItem()

int getSelectedIndex()

getItem() - returns a string containing the name of the item.

getSelectedIndex() - returns the index of the item. The first item is at

index 0. **getItemCount() :** To obtain the number of items in the list

Syntax : int getItemCount()

select() : currently selected item with either a zero-based integer index or a string that will match a name in the list.

Syntax :

void select(int *index*)

void select(String
name)

getItem() : we can obtain the name associated with the item at that index.

Syntax : String getItem(int *index*)

here, *index* specifies the index of the desired item.

Handling Choice controls:

- Each time a choice is selected, an item event is generated. This is sent to **ItemListener** that previously registered to receive an item event notifications from that component.
- **ItemListener** interface defines the **itemStateChanged()** method.
- An **ItemEvent** object is supplied as the argument to this method. It contains information about the event

EXAMPLE PROGRAM:

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

```
import java.applet.*;
```

```
import
```

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

```
/*
```

```
<applet code=choice width=1000 height=1000>
```

```
</applet>
```

```
*/
```

```
public class choice extends Applet implements
```

```
ItemListener { String msg;
```

```
Choice branch;
```

```
public void init()
```

```
{ Label l1=new
```

```
Label("Branch");
```

```
branch=new Choice();
```

```
branch.add("CSE");
```

```
branch.add("ECE");
```

```
branch.add("EEE");
```

```
add(l1);
```

```
add(branch);
```

```
branch.addItemListener(this);
```

```
}
```

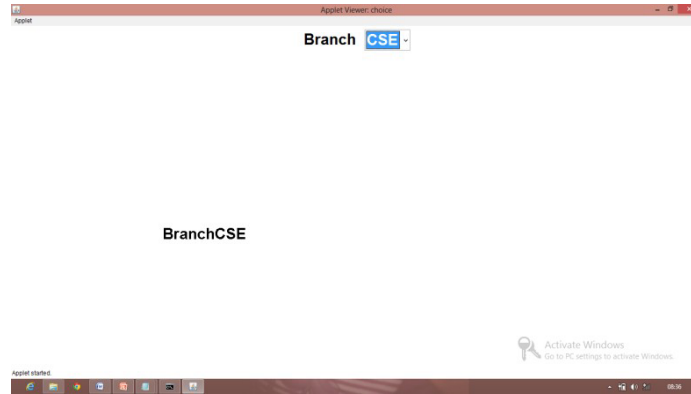
```
public void itemStateChanged(ItemEvent ie)
```

```
        { repaint();  
    }  
    public void paint(Graphics  
g){ msg="Branch"+branch.getSelectedItem();
```

```

    rawString(msg,300,420);
}
}

```



List

- The **List** class is used to create a pop-up list.
- **Choice** object, which shows only the single selected item in the menu,
- **List** object can be constructed to show any number of choices in the visible window
- From list it is possible to select more than one item.
- It provides multiple-choice, scrolling selection list.
- It can also be created to allow multiple selections.

List provides these constructors:

List()

List(int *numRows*)

List(int *numRows*, boolean *multipleSelect*)

- The first version creates a **List** control that allows only one item to be selected at any one time.
- In the second form, the value of *numRows* specifies the number of entries in the list that will always be visible .
- In the third form, if *multipleSelect* is **true**, then the user may select two or more items at a time. If it is **false**, then only one item may be selected.

add():

It is used to add items to the list

Syntax:

void add(String *name*)

void add(String *name*, int *index*)

Here, *name* is the name of the item added to the list. The first form adds items to the end of the list. The second form adds the item at the index specified by *index*. Indexing begins at zero.

getSelectedItem() :used to get the name of selected item.

getSelectedIndex():used to get the selected item index.

Syntax :

String

getSelectedItem() int

getSelectedIndex()

getSelectedItems() or getSelectedIndexes() :

Syntax :

String[]

getSelectedItems() int[]

getSelectedIndexes()

getSelectedItems() - returns an array containing the names of the currently selected items.

getSelectedIndexes() - returns an array containing the indexes of the currently selected items.

getItemCount() : It obtain the number of items in the list

Syntax :

int getItemCount()

getItem() : Given an index, we can obtain the name associated with the item at that index

Syntax : String getItem(int *index*)

here, *index* specifies the index of the desired item.

Handling Lists:

- Each time a item is selected in a list, an item event is generated. This is sent to **ItemListener** that previously registered to receive an item event notifications from that component.
- **ItemListener** interface defines the **itemStateChanged()** method.
- An **ItemEvent** object is supplied as the argument to this method. It contains information about the event

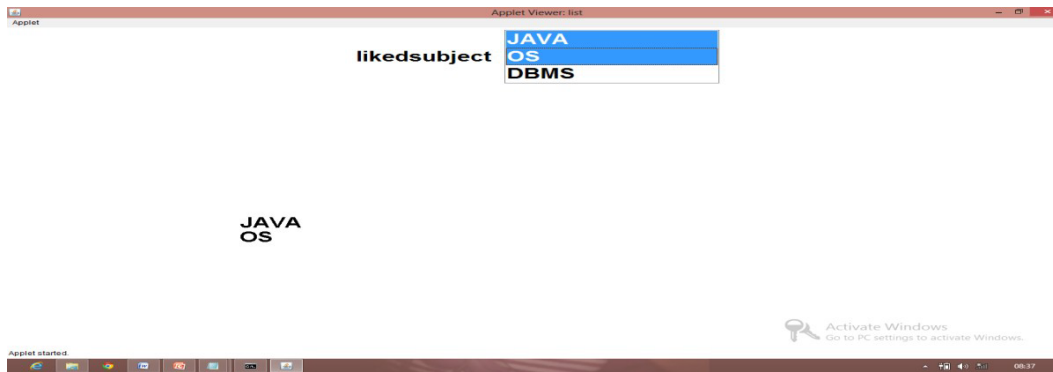
Example program:

```
import java.awt.*;
import
java.applet.*;
import java.awt.event.*;
/*
<applet code=list width=1000 height=1000>
</applet>
*/
public class list extends Applet implements
    ItemListener { String msg;
    List fav;
    public void init()
    {
        Label l1=new
        Label("likedsubject"); fav=new
        List(3,true); fav.add("JAVA");
        fav.add("OS");
        fav.add("DBMS");
add(l1);
add(fav);
        fav.addItemListener(this);
    }

    public void itemStateChanged(ItemEvent ie)    {
        repaint();
    }

    public void paint(Graphics g)
    { String idx[];
    msg="Current:";
    idx=fav.getSelectedItems
    (); msg=idx[0];
    g.drawString(msg,300,42
    0); msg=idx[1];
    g.drawString(msg,300,45
    0); msg=idx[2];
    g.drawString(msg,300,48
```

```
0);  
}  
}
```

TextFields

- The **TextField** class implements a single-line text-entry area, usually called an *edit control*.
- Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections.
- **TextField** is a subclass of **TextComponent**.

TextField defines the following constructors:

```
TextField()
TextField(int
numChars)
TextField(String str)
TextField(String str, int numChars)
```

- The first version creates a default text field.
- The second form creates a text field that is *numChars* characters wide.
- The third form initializes the text field with the string contained in *str*.
- The fourth form initializes a text field and sets its width.

Methods:

getText() : To obtain the string currently contained in the text field

setText() : To set the text.

Syntax :

```
String getText()
void setText(String
str) here, str is the new
string.
```

select() : The user can select a portion of the text in a text field. Also, we can select a portion of text under program control

getSelectedText() : our program can obtain the currently selected text Syntax :

```
String getSelectedText()
void select(int startIndex, int
endIndex) getSelectedText() returns the
selected text.
```

The **select()** method selects the characters beginning at *startIndex* and ending at *endIndex-1*. **setEditable()** : we can control whether the contents of a text field may be modified by the user or not. **isEditable()** : we can determine editability

Syntax :

boolean isEditable()

void setEditable(boolean *canEdit*)

isEditable() returns true if the text may be changed and false if not.

In **setEditable()**, if *canEdit* is true, the text may be changed. If it is false, the text cannot be altered.
setEchoChar() :

- There may be times when we will want the user to enter text that is not displayed, such as a password.

We can disable the echoing of the characters as they are typed.

- This method specifies a single character that the TextField will display when characters are entered (thus, the actual characters typed will not be shown and we can display like * or \$).

echoCharIsSet() : we can check a text field to see if it is in echoChar mode.

getEchoChar() : we can retrieve the echo character.

Syntax :

```
void setEchoChar(char  
ch) boolean  
echoCharIsSet() char  
getEchoChar()
```

here, *ch* specifies the character to be echoed.

Handling a TextField

- Since text fields perform their own editing functions, your program generally will not respond to individual key events that occur within a text field.
- However, you may want to respond when the user presses ENTER. When this occurs, an action event is generated.

TextArea :

- Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor that is textarea.
- In text area you may enter more than one line of text.

Constructors :

```
TextArea()  
TextArea(int numLines, int  
numChars) TextArea(String str)  
TextArea(String str, int numLines, int numChars)  
TextArea(String str, int numLines, int numChars, int  
sBars)
```

here, *numLines* → the height, in lines, of
the text area, *numChars* → its
width, in characters. *str* → Initial
text.

It supports the **getText()**, **setText()**, **getSelectedText()**, **select()**, **isEditable()**, and **setEditable()** methods.

```
import java.awt.*;  
import java.applet.*;  
import  
java.awt.event.*;  
/*  
<applet code="text" width=1000 height=1000>  
</applet>  
*/
```

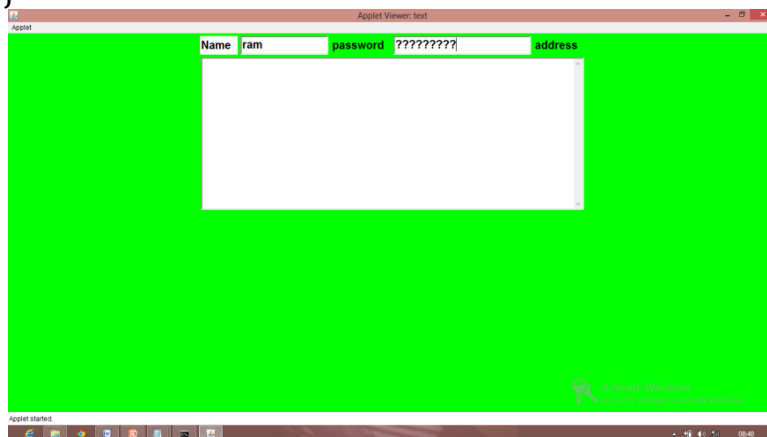
```
public class text extends Applet implements  
ActionListener { TextField t1,t2;
```

```
TextArea ta;  
String  
name,pass,address;  
public void init() {  
Label l1=new Label("Name");  
t1=new TextField(12);
```

```

add(l1); add(t1);
setBackground(Color.green);
Label l2=new
Label("password"); t2=new
TextField(20);
add(l2); add(t2);
Label l3=new
Label("address"); ta=new
TextArea();
add(l3); add(ta);
//t1.setEditable(false);
t2.setEchoChar('?');
t1.addActionListener(thi
s);
t2.addActionListener(thi
s);
}
public void actionPerformed(ActionEvent
ae) { repaint();
}
public void paint(Graphics g)
{ name=t1.getText();
g.drawString(name,300,300);
pass=t2.getText();
g.drawString(pass,300,330);
address=ta.getText();
g.drawString(address,300,3
60);
}
}

```



Scroll bars

- *Scroll bars* are used to select continuous values between a specified minimum and maximum.
- Scroll bars may be oriented horizontally or vertically.
- A scroll bar is actually a composite of several individual parts.
- Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow.
- The current value of the scroll bar relative to its minimum and maximum values is indicated by the *slider box* (or *thumb*) for the scroll bar.

- The slider box can be dragged by the user to a new position.
- The scroll bar will then reflect this value.
- Scroll bars are encapsulated by the **Scrollbar** class.

Scrollbar defines the following constructors:

```
Scrollbar( )
Scrollbar(int
style)
Scrollbar(int style, int initialValue, int thumbSize, int min, int max)
```

The first form creates a vertical scroll bar.

- The second and third forms allow you to specify the orientation of the scroll bar.
- If *style* is **Scrollbar.VERTICAL**, a vertical scroll bar is created.
- If *style* is **Scrollbar.HORIZONTAL**, the scroll bar is horizontal.
- In the third form of the constructor, the initial value of the scroll bar is passed in *initialValue*. The number of units represented by the height of the thumb is passed in *thumbSize*. The minimum and maximum values for the scroll bar are specified by *min* and *max*

setValues() : If we construct a scroll bar by using one of the first two constructors, then we need to set its parameters

Syntax :

```
void setValues(int initialValue, int thumbSize, int min, int max)
```

getValue() : To obtain the current value of the scroll bar, returns the current setting.

setValue() : To set the current value

Syntax :

```
int getValue( )
void setValue(int newValue)
```

Here, *newValue* specifies the new value for the scroll bar.

Handling Scroll Bars :

- Each time a user interacts with a scroll bar, an AdjustmentEvent object is generated.
- This is sent to any AdjustmentListener that previously registered to receiving adjustment event notifications from that component.
- To process scroll bar events, we need to implement the AdjustmentListener interface..
- AdjustmentListener interface have adjustmentValueChanged() method.
- An AdjustmentEvent object is supplied as the argument to this

```
method. import java.awt.*;
import java.applet.*;
import
java.awt.event.*;
/*
<applet code="sbar" width=1000 height=1000>
</applet>
*/
```

```
public class sbar extends Applet implements
AdjustmentListener { String msg="";
Scrollbar vsb,hsb;
```

```
    public void init()          {
        vsb=new Scrollbar(Scrollbar.VERTICAL,0,1,0,100);
        hsb=new
        Scrollbar(Scrollbar.HORIZONTAL,0,1,0,1000);
        add(vsb);
        add(hsb);
        vsb.addAdjustmentListener(th
```

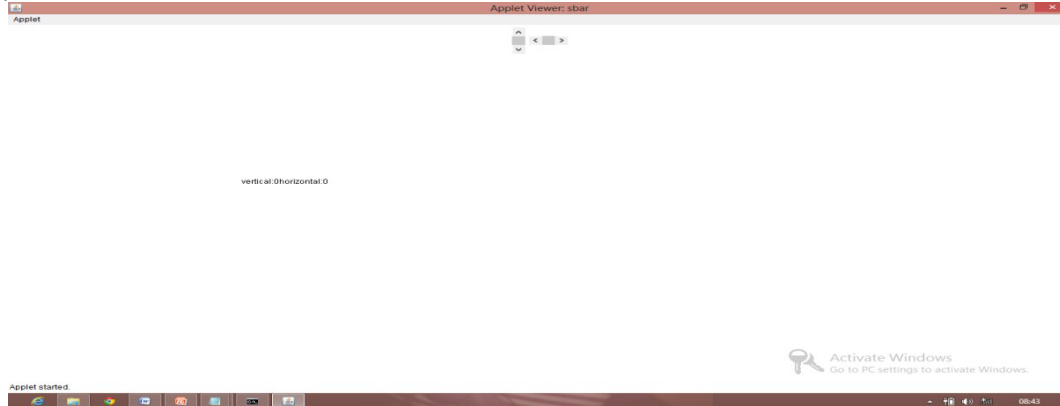
```
is);  
hsb.addAdjustmentListener(th  
is);  
    }  
public void adjustmentValueChanged(AdjustmentEvent ae)    {
```



```

        repaint();
    }
    public void paint(Graphics g)
    { msg="vertical:"+vsb.getValue();
      msg+="horizontal:"+hsb.getValue
    (); g.drawString(msg,300,300);
    }
}

```



JSplitPane

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

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

Nested Class

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

Useful Fields

Modifier and Type	Field	Description
static String	BOTTOM	It use to add a Component below the other Component.
static String	CONTINUOUS_LAYOUT_PROPERTY	Bound property name for continuousLayout.
static String	DIVIDER	It uses to add a Component that will represent the divider.
static int	HORIZONTAL_SPLIT	Horizontal split indicates the Components are split along the x axis.
protected int	lastDividerLocation	Previous location of the split pane.

protected Component	leftComponent	The left or top component.
static int	VERTICAL_SPLIT	Vertical split indicates the Components are split along the y axis.
protected Component	rightComponent	The right or bottom component.
protected int	orientation	How the views are split.

Constructors

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

Useful Methods

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

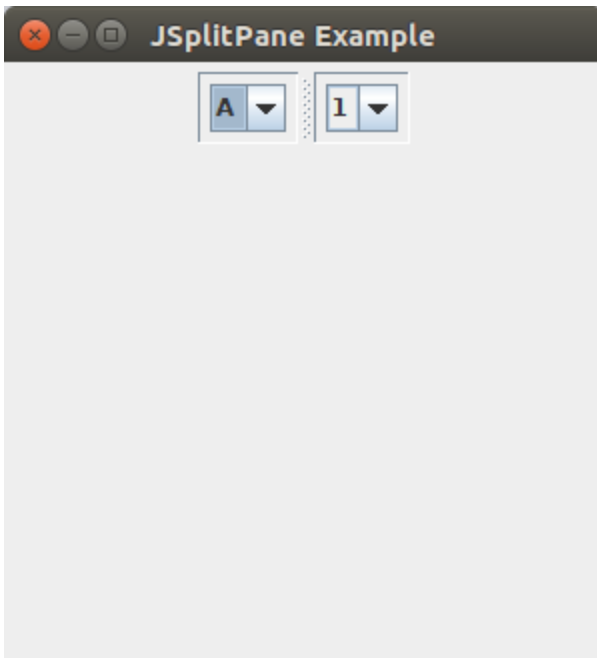
SplitPaneUI	getUI()	It returns the SplitPaneUI that is providing the current look and feel.
boolean	isContinuousLayout()	It gets the continuousLayout property.
boolean	isOneTouchExpandable()	It gets the oneTouchExpandable property.
void	setOrientation(int orientation)	It gets the orientation, or how the splitter is divided.

JSplitPane Example

```
import
java.awt.FlowLayout;
import java.awt.Panel;
import
javax.swing.JComboBox;
import javax.swing.JFrame;
import
javax.swing.JSplitPane;
public class
JSplitPaneExample {
private static void createAndShow() {
// Create and set up the window.
final JFrame frame = new JFrame("JSplitPane Example");
// Display the window.
frame.setSize(300,
300);
frame.setVisible(true);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// set flow layout for the frame
frame.getContentPane().setLayout(new
FlowLayout()); String[] option1 = { "A","B","C","D","E"
};
JComboBox box1 = new
JComboBox(option1); String[] option2 =
{"1","2","3","4","5"};
JComboBox box2 = new
JComboBox(option2); Panel panel1 = new
Panel(); panel1.add(box1);
Panel panel2 = new
Panel();
panel2.add(box2);
JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, panel1, panel2);
// JSplitPane splitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
// panel1, panel2);
frame.getContentPane().add(splitPa
ne);
}
public static void main(String[] args) {
// Schedule a job for the event-dispatching thread:
// creating and showing this application's GUI.
javax.swing.SwingUtilities.invokeLater(new
```

```
Runnable() {  
    public void run()  
        { createAndSho  
          w();  
        }  
    }  
};  
}
```

Output:



Layout Managers

- Layout manager automatically arranges components within a window by using some type of algorithm.
 - Each Container object has a layout manager associated with it.
 - 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.
- setLayout():** it is used to set new layout.

Syntax :

```
void setLayout(LayoutManager layoutObj)
here, layoutObj is a reference to the desired layout manager.
```

Ex: `setLayout(new BorderLayout());`

- Some of the Layout – Managers are :
 1. Flow Layout
 2. Border Layout
 3. Grid Layout
 4. GridBag Layout
 5. Card Layout

FlowLayout

- It is the default layout manager. .
- It implements a simple layout style, which is similar to how words are flow in a text editor.
- In FlowLayout Components are laid out from left to right and top to bottom.

When no more components fit on a line, the next one appears on the next line. A small space is left between each component, above and below, as well as left and right

Constructors

`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 us specify how each line is aligned. Valid values for *how* are as follows: `FlowLayout.LEFT`, `FlowLayout.CENTER`, `FlowLayout.RIGHT`


These values specify left, center, and right alignment, respectively.

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

```
import java.awt.*;
import java.applet.*;
import
java.awt.event.*;
/*
<applet code="flowlayout" width=500 height=500>
</applet>
*/

public class flowlayout extends
    Applet { Label l1,l2,l3;

    public void init()      {
        setLayout(new FlowLayout(FlowLayout.CENTER));
        l1=new Label("Enter a value");
        add(l1);
        l2=new Label("enter b
value"); add(l2);
        l3=new
        Label("Result");
        add(l3);
    }
}
```




BorderLayout

A border layout lays out a container, arranging and resizing its components to fit in to five regions It has

- 4 narrow, fixed-width components at the edges &
- 1 large area in the center.

- The 4 sides are referred to as north, south, east, west. The middle area is called the center.

Constructors:

```
BorderLayout()
BorderLayout(int horz, int
vert)
```

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

Constants :

- The following 5 constants that specify the regions:
BorderLayout.CENTER

BorderLayout.SOUTH BorderLayout.EAST
BorderLayout.WEST
BorderLayout.NORTH

Methods

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

Syntax :

void add(Component *compObj*, Object *region*); here, *compObj* is the component to be added

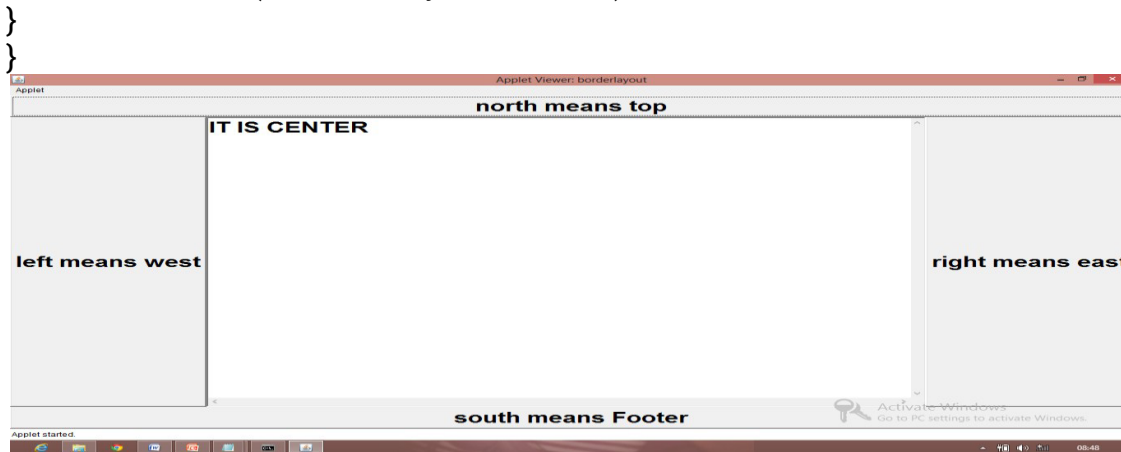
region specifies where the component will be added.

Example Program:

```
import java.awt.*;
import java.applet.*;
import
java.awt.event.*;
/*
<applet code="borderlayout" width=1000 height=1000>
</applet>
*/
public class borderlayout extends Applet

{ Button t,l,r,f;
```

```
    public void init()    {
        t=new Button("north means top");
        setLayout(new BorderLayout());
        add(t,BorderLayout.NORTH);
        Button f=new Button("south means Footer");
        add(f,BorderLayout.SOUTH);
        r=new Button("right means
        east");
        add(r,BorderLayout.EAST);
        l=new Button("left means
        west");
        add(l,BorderLayout.WEST);
        String msg="IT IS CENTER";
        TextArea ta=new
        TextArea(msg);
        add(ta,BorderLayout.CENTER);
```

**GridLayout :**

- GridLayout lays out components in a two-dimensional grid.
- When we instantiate a GridLayout, we define the number of rows and columns.

constructors :

```
GridLayout( )
GridLayout(int numRows, int numColumns )
```

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

- First form creates grid layout with one column per component in a single row.
- Second form creates a grid layout with the specified number of rows and columns.

- Third form allows us to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively

Source program:

```
import java.awt.*;
import java.applet.*;
import
java.awt.event.*;
/*
<applet code="grid" width=1000 height=1000>
</applet>
*/
public class grid extends
Applet { Button
b1,b2,b3,b4,b5,b6;
public void init()
{
    setLayout(new
    GridLayout(2,3)); b1=new
    Button("First");
    b2=new
    Button("second");
    b3=new Button("third");
    b4=new Button("fourth");
    b5=new Button("five");
    b6=new Button("six");
    add(b1);
    add(b2);
    add(b3);
    add(b4);
    add(b5);
    add(b6);
}
}
```



GridBagLayout:

- More flexible than Grid Layout.
- It allows the components span over multiple rows and columns.
- It also enables us to resize the components by assigning appropriate weights to them.
- grid bag useful is you can specify the relative placement of components by specifying their positions within cells inside a grid.

- The key to the grid bag is that each component can be a different size, and each row in the grid can have a different number of columns.
- This is why the layout is called a *grid bag*. It's a collection of small grids joined together
- The location and size of each component in a grid bag are determined by a set of constraints.
- The constraints are contained in an object of type **GridBagConstraints**.

- Constraints include the height and width of a cell, and the placement of a component, its alignment, and its anchor point within the cell.

Procedure to set GridBagLayout:

- The general procedure for using a grid bag is to first create a new **GridBagLayout** object and to make it the current layout manager.
- Then, set the constraints that apply to each component that will be added to the grid bag. Finally, add the components to the layout manager

GridBagLayout defines only one constructor, which is shown here:

```
GridBagLayout()
```

Methods:

There is one method, however, that you must use: **setConstraints()**. It is

shown here: `void setConstraints(Component comp, GridBagConstraints cons)`

Here, *comp* is the component for which the constraints specified by *cons* apply. This method sets the constraints that apply to each component in the grid bag.

The key to successfully using **GridBagLayout** is the proper setting of the constraints, which are stored in a

GridBagConstraints object. **GridBagConstraints** defines several fields that you

Constructors of GridBagConstraints class :

```
GridBagConstraints()
```

```
GridBagConstraints(int gridx, int gridy, int gridwidth, int gridheight,  
int weightx, int weighty, int anchor, int fill, Insets insets, int ipadx, int
```

```
ipady)
```

Data Members of GridBagConstraints class

Data Member Purpose

gridx to specify the row position in the upper-left region of the component's display

area. Grid to specify the column position in the upper-left region of the component's display area.

Gridwidth to specify the number of columns in the display area of the component. The defaultvalue is 1 pixel.

gridheight to specify the number of rows in the display area of the component. The defaultvalue is 1 pixel.

Weightx to specify the horizontal stretch of the component to fill the display area of the container. The default value is 0.

Gridy to specify the vertical stretch of the component to fill the display area of the container. The default value is 0.

Anchor to specify the position of a component in a display area, when the component is smaller than the display area.

Valid values of Anchor Data Member :

Value of Anchor

Placement of Component

GridBagConstraints.CENTER

Center of Component

GridBagConstraints.NORTH

North part of Container

GridBagConstraints.NORTHEAST

north-east part of

Container GridBagConstraints.NORTHWEST

north-west part of

Container GridBagConstraints.SOUTH

south part of Container

GridBagConstraints.SOUTHEAST

south-east part of

Container GridBagConstraints.SOUTHWEST

south-west part of

Container GridBagConstraints.EAST

east part of Container

GridBagConstraints.WEST

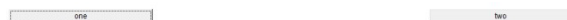
west part of Con

Example Program:


```

import java.awt.*;
import
java.awt.event.*;
import java.applet.*;
/*
<applet code="GridBagDemo" width=500 height=500>
</applet>
*/
public class GridBagDemo extends Applet
{ String msg = "";
  Button b1,b2;
  public void init()
  {
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);
    b1=new
    Button("one");
    b2=new
    Button("two");
    gbc.weightx = 1.0;
    gbc.ipadx = 200;
    gbc.insets = new Insets(4, 4, 0, 0);
    gbc.anchor =
    GridBagConstraints.WEST;
    gbc.gridwidth =
    GridBagConstraints.RELATIVE;
    gbag.setConstraints(b1, gbc);
    gbc.gridwidth =
    GridBagConstraints.REMAINDER;
    gbag.setConstraints(b2, gbc);
    add(b1);
    add(b2);
  }
}

```



Card Layout:

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.

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*. The following example creates a two-level card deck that allows the user to select an operating system. Windows-based operating systems are displayed in one card. Macintosh and Solaris are displayed in the other card.

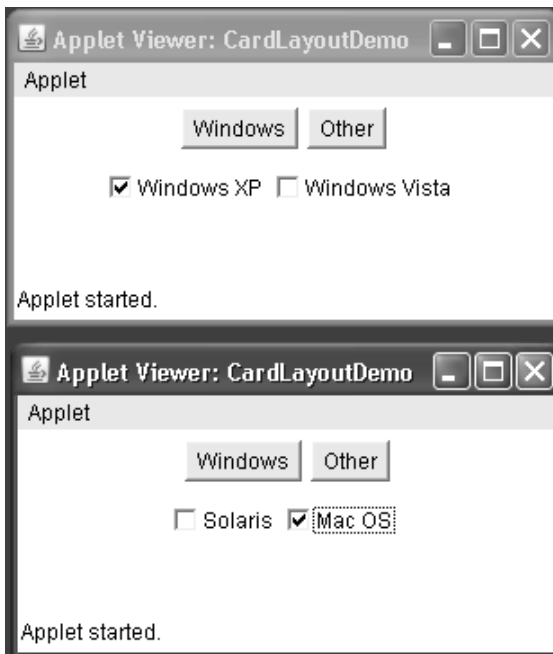
```
// Demonstrate  
CardLayout. import  
java.awt.*;  
import java.awt.event.*;  
  
import java.applet.*;  
/*  
<applet code="CardLayoutDemo" width=300 height=100>  
</applet>  
*/  
public class CardLayoutDemo extends Applet implements ActionListener,  
MouseListener { Checkbox winXP, winVista, solaris, mac;  
Panel osCards;  
CardLayout cardLO;  
Button Win, Other;  
public void init() {  
Win = new Button("Windows");  
Other = new Button("Other");  
add(Win);
```

```
add(Other);  
cardLO = new CardLayout();  
osCards = new Panel();  
osCards.setLayout(cardLO); // set panel layout to card layout
```

```

winXP = new Checkbox("Windows XP", null,
true); winVista = new Checkbox("Windows
Vista"); solaris = new Checkbox("Solaris");
mac = new Checkbox("Mac OS");
// add Windows check boxes to a
panel Panel winPan = new Panel();
winPan.add(winXP);
winPan.add(winVista);
// add other OS check boxes to a
panel Panel otherPan = new
Panel(); otherPan.add(solaris);
otherPan.add(mac);
// add panels to card deck panel
osCards.add(winPan, "Windows");
osCards.add(otherPan, "Other");
// add cards to main applet
panel add(osCards);
// register to receive action
events
Win.addActionListener(this);
Other.addActionListener(this);
// register mouse
events
addMouseListener(thi
s);
}
// Cycle through panels.
public void mousePressed(MouseEvent me)
{ cardLO.next(osCards);
}
// Provide empty implementations for the other MouseListener
methods. public void mouseClicked(MouseEvent me) {
}
public void mouseEntered(MouseEvent me) {
}
public void mouseExited(MouseEvent me) {
}
public void mouseReleased(MouseEvent me) {
}
public void actionPerformed(ActionEvent
ae) { if(ae.getSource() == Win)
{ cardLO.show(osCards, "Windows");
}
else {
cardLO.show(osCards, "Other");
}
}
}
}

```



Following example illustrates `abe`, `Button`, `Choice`, `List`, `Checkboxes`, `TextFields`, `TextArea`

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/*
<applet code="student" width=1000 height=1000>
</applet>
*/
public class student extends Applet implements ActionListener,
    ItemListener { TextField t1,t2,t3;
        TextArea ta;
        String sname,sno,mail,address,bra,gender;
        Label l1,l2,l3,l4;          Button a,d;          Checkbox m,f;
        CheckboxGroup cbg;          Choice branch;
    public void init()
    { setBackground(Color.green);
      setForeground(Color.blue);
      l1=new Label("studentname");
      t1=new TextField(12);
      add(l1);          add(t1);
      l2=new Label("studentno");
      t2=new TextField(12);
      add(l2);          add(t2);
      l3=new Label("mail");
      t3=new TextField(12);
      add(l3);          add(t3);
      l4=new Label("Address");
      ta=new TextArea(10,20);
      add(l4);          add(ta);
      Label l4=new Label("gender");
      cbg=new CheckboxGroup();
```

```

m=new Checkbox("Male",cbg,true);
f=new Checkbox("Female",cbg,false);
add(l4);      add(m);      add(f);
branch=new Choice();
branch.add("CSE");      branch.add("ECE");
branch.add("EEE");
add(branch);
a=new Button("save and display");
add(a);
a.addActionListener(this);
}
public void actionPerformed(ActionEvent ae)
{ repaint();
}

public void paint(Graphics g)
{ sname = t1.getText();
sname="studentname:"+sname;
sno = t2.getText();
sno="studentno:"+sno;
mail = t3.getText();
mail="studentmail:"+mail;
String addr=ta.getText();
address="Address:"+addr;
gender="Gender:"+cbg.getSelectedCheckbox().getLabel();
bra="Branch"+branch.getSelectedItem();
g.drawString(sname,300,300);
g.drawString(sno,300,330);
g.drawString(mail,300,360);
g.drawString(address,300,390)
;
g.drawString(gender,300,420);
g.drawString(bra,300,450);
}
}

```



1. Develop an Applet program to accept two numbers from user and output the sum, difference in the respective text boxes.

Source program:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/*
<applet code="addsub" width=500 height=500>
</applet>
*/
public class addsub extends Applet implements ActionListener
{
    TextField t1,t2,t3;
    Label l1,l2,l3;      Button a,d;

    public void init()      {
        l1=new Label("Enter a value");
        t1=new TextField(12);      add(l1);
        add(t1);      l2=new Label("enter b value");
        t2=new TextField(12);      add(l2);      add(t2);
        l3=new Label("Result");      t3=new
        TextField(12); add(l3);      add(t3);
        a=new Button("Add");      d=new Button("Diff");
        add(a);      add(d);
        a.addActionListener(this);
        d.addActionListener(this);
    }
    public void actionPerformed(ActionEvent ae)
    {
        String n1 = t1.getText();
        String n2 =
        t2.getText(); int a=
        Integer.parseInt(n1);
        int b= Integer.parseInt(n2);
        String name=ae.getActionCommand();//it gets the label on
        putton if(name.equals("Add"))
        {
            int sum=a+b;
            t3.setText(Integer.toString(sum)
            );
        }
        else
```


{

```
        int diff=a-b;  
        t3.setText(Integer.toString(diff)  
        );  
    }  
  
}
```

Swing:

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

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

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 apperance is independent of internal representation.

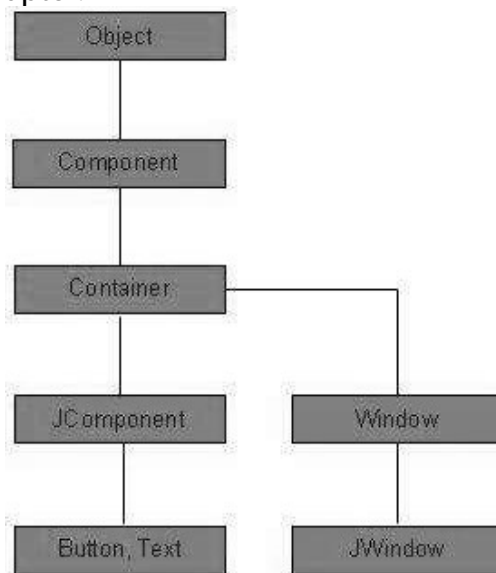
Pluggable look-and-feel - SWING based GUI Application look and feel can be changed at run-time, based on available values.

Swing – Controls

Every user interface considers the following three main aspects:

UI Elements: These are the core visual elements the user eventually sees and interacts with. GWT provides a huge list of widely used and common elements varying from basic to complex, which we will cover in this tutorial. **Layouts:** They define how UI elements should be organized on the screen and provide a final look and feel to the GUI (Graphical User Interface). This part will be covered in the Layout chapter.

Behavior: These are the events which occur when the user interacts with UI elements. This part will be covered in the Event Handling chapter.



Every SWING controls inherits properties from the following Component class hierarchy.

Sr. No.	Class & Description
1	Component A Component is the abstract base class for the non menu user-interface controls of SWING. Component represents an object with graphical representation

2	Container A Container is a component that can contain other SWING components
---	--

3	JComponent A JComponent is a base class for all SWING UI components. In order to use a SWING component that inherits from JComponent, the component must be in a containment hierarchy whose root is a top-level SWING container
---	--

Component Class:

Introduction

The class **Component** is the abstract base class for the non-menu user-interface controls of AWT. Component represents an object with graphical representation.

Field:

Following are the fields for **java.awt.Component** class:

static float BOTTOM_ALIGNMENT - Ease-of-use constant for getAlignmentY.

static float CENTER_ALIGNMENT - Ease-of-use constant for getAlignmentY and getAlignmentX.

static float LEFT_ALIGNMENT - Ease-of-use constant for getAlignmentX.

static float RIGHT_ALIGNMENT - Ease-of-use constant for

getAlignmentX. **static float TOP_ALIGNMENT** - Ease-of-use constant for getAlignmentY().

Class Constructors:

s.no	Constructor & Description
1	protected Component() This creates a new Component

Class Methods:

Sr.No.	Method & Description
1	void add(PopupMenu popup) Adds the specified popup menu to the component.
2	void addComponentListener(ComponentListener l) Adds the specified component listener to receive the component events from this component.

Container Class

Introduction

The class **Container** is the super class for the containers of AWT. Container object can contain other AWT components.

Class Constructors

S.no	Constructor & Description
1	Container() This creates a new Container.

Class Methods:

S.No.	Method & Description
1	Component add(Component comp) Appends the specified component to the end of this container.
2	Component add(Component comp, int index) Adds the specified component to this container at the given position.
3	void add(Component comp, Object constraints) Adds the specified component to the end of this

	container.
4	void add(Component comp, Object constraints, int index) Adds the specified component to this container with the specified constraints at the specified index.

JComponent Class

Introduction

The class **JComponent** is the base class for all Swing components except top-level containers. To use a component that inherits from JComponent, you must place the component in a containment hierarchy, whose root is a top-level SWING container.

Field:

Following are the fields for **java.awt.Component** class:

protected AccessibleContext accessibleContext - The AccessibleContext associated with this JComponent.

protected EventListenerList listenerList - A list of event listeners for this component.

static String TOOL_TIP_TEXT_KEY - The comment to display when the cursor is over the component, also known as a "value tip", "flyover help", or "flyover label".

protected ComponentUI ui - The look and feel delegate for this component.

static int UNDEFINED_CONDITION - Constant used by some of the APIs to mean that no condition is defined.

Class Constructors:

s.no	Constructor & Description
1	JComponent() Default JComponent constructor.

Class Methods:

Sr.No.	Method & Description
1	float getAlignmentX() Overrides Container.getAlignmentX to return the vertical alignment.
2	float getAlignmentY() Overrides Container.getAlignmentY to return the horizontal alignment.
3	void addNotify() Notifies this component that it now has a parent component.
4	boolean contains(int x, int y) Gives the UI delegate an opportunity to define the precise shape of this component for the sake of mouse processing.

SWING UI Elements

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

Sr.No.	Control & Description
1	JLabel A JLabel object is a component for placing text in a container.
2	JButton This class creates a labeled button.
3	JCheckBox A JCheckBox is a graphical component that can be in either an on (true) or off (false) state.
4	JRadioButton The JRadioButton class is a graphical component that can be in either an on (true) or off (false) state. in a group.
5	JList A JList component presents the user with a scrolling list of text items.
6	JTextField A JTextField object is a text component that allows editing of a single line of text.
7	JTextArea A JTextArea object is a text component that allows editing of a multiple lines of text.
8	JScrollbar A Scrollbar control represents a scroll bar component in order to enable the user to select from a range of values.

JLabel Class:

Introduction

The class **JLabel** can display either text, an image, or both. Label's contents are aligned by setting the vertical and horizontal alignment in its display area. By default, labels are vertically centered in their display area. Text- only labels are leading edge aligned, by default; image-only labels are horizontally centered, by default.

Field

Following are the fields for **javax.swing.JLabel** class:

- **protected Component labelFor**

Class Constructors

s.no	Constructor & Description
1	JLabel() Creates a JLabel instance with no image and with an empty string for the title.
2	JLabel(Icon image) Creates a JLabel instance with the specified image.
3	JLabel(Icon image, int horizontalAlignment) Creates a JLabel instance with the specified image and horizontal alignment.
4	JLabel(String text) Creates a JLabel instance with the specified text.
5	JLabel(String text, Icon icon, int horizontalAlignment) Creates a JLabel instance with the specified text, image, and horizontal alignment.
6	JLabel(String text, int horizontalAlignment) Creates a JLabel instance with the specified text and horizontal alignment.

Class Methods

Sr.No.	Method & Description
1	String getText() Returns the text string that the label displays.
2	int getHorizontalAlignment() Returns the alignment of the label's contents along the X axis.
3	int getHorizontalTextPosition() Returns the horizontal position of the label's text, relative to its image.

JButton

Class

Introduction

The class **JButton** is an implementation of a push button. This component has a label and generates an event when pressed. It can also have an Image.

Class Constructors

S.No	Constructor & Description
1	JButton() Creates a button with no set text or icon.
2	JButton(Action a) Creates a button where properties are taken from the Action supplied.
3	JButton(Icon icon) Creates a button with an icon.
4	JButton(String text) Creates a button with the text.
5	JButton(String text, Icon icon) Creates a button with an initial text and an icon.

JCheckBox

Class

Introduction

The class **JCheckBox** is an implementation of a checkbox - an item that can be selected or deselected, and which displays its state to the user.

Field

Following are the fields for **javax.swing.JCheckBox** class.

static String BORDER_PAINTED_FLAT_CHANGED_PROPERTY – Identifies a change to the flat property.

Class Constructors

S.No	Constructor & Description
1	JCheckBox() Creates an initially unselected checkbox button with no text and no icon.
2	JCheckBox(Action a) Creates a checkbox where the properties are taken from the Action supplied.

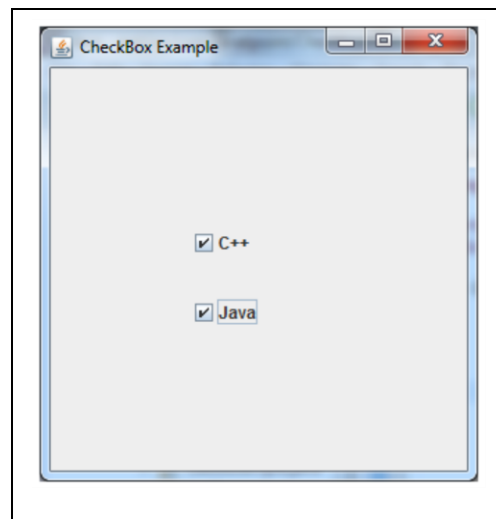
3	JCheckBox(icon icon) Creates an initially unselected checkbox with an icon.
---	--

4	JCheckBox(Icon icon, boolean selected) Creates a checkbox with an icon and specifies whether or not it is initially selected.
5	JCheckBox(String text) Creates an initially unselected checkbox with text.
6	JCheckBox(String text, boolean selected) Creates a checkbox with the text and specifies whether or not it is initially selected.
7	JCheckBox(String text, Icon icon) Creates an initially unselected checkbox with the specified text and icon.
8	JCheckBox(String text, Icon icon, boolean selected) Creates a checkbox with text and icon, and specifies whether or not it is initially selected.

Class Methods

S.No	Method & Description
1	void updateUI() Resets the UI property to a value from the current look and feel.
2	protected String paramString() Returns a string representation of this JCheckBox.

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



JRadioButton

Class Introduction

The class **JRadioButton** is an implementation of a radio button - an item that can be selected or deselected, and which displays its state to the user.

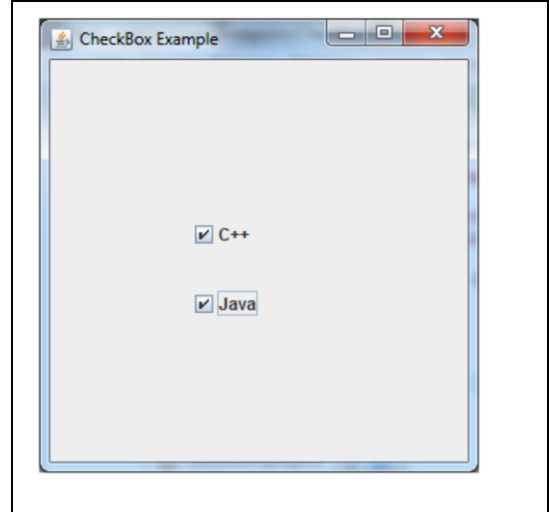
Class Constructors

S.No	Constructor & Description
1	JRadioButton() Creates an initially unselected radio button with no set text.
2	JRadioButton(Action a) Creates a radiobutton where properties are taken from the Action supplied.
3	JRadioButton(Icon icon) Creates an initially unselected radio button with the specified image but no text.
4	JRadioButton(Icon icon, boolean selected) Creates a radio button with the specified image and selection state, but no text.
5	JRadioButton(String text, boolean selected) Creates a radio button with the specified text and selection state.

6	JRadioButton(String text, Icon icon) Creates a radio button that has the specified text and image, and which is initially unselected.
---	--

7	JRadioButton(String text, Icon icon, boolean selected) Creates a radio button that has the specified text, image, and selection state.
---	--

```
import javax.swing.*;
public class RadioButtonExample
{ JFrame f;
  RadioButtonExample(){ f=new
  JFrame();
  JRadioButton r1=new JRadioButton("A) Male");
  JRadioButton r2=new JRadioButton("B) Female");
  r1.setBounds(75,50,100,30);
  r2.setBounds(75,100,100,30);
  ButtonGroup bg=new ButtonGroup();
  bg.add(r1);bg.add(r2);
  f.add(r1);f.add(r2);
  f.setSize(300,300);
  f.setLayout(null);
  f.setVisible(true);
}
public static void main(String[] args) {
    new RadioButtonExample();
}
}
```



JList Class

Introduction

The class **JList** is a component which displays a list of objects and allows the user to select one or more items. A separate model, **ListModel**, maintains the contents of the list.

Field

Following are the fields for **javax.swing.JList** class –

- **static int HORIZONTAL_WRAP** – Indicates a "newspaper style" layout with cells flowing horizontally then vertically.
- **static int VERTICAL** – Indicates a vertical layout of cells, in a single column; the default layout.
- **static int VERTICAL_WRAP** – Indicates a "newspaper style" layout with cells flowing vertically then horizontally.

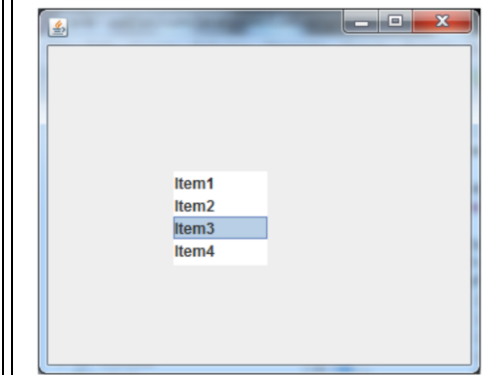
Class Constructors

S.No	Constructor & Description
1	JList() Constructs a JList with an empty, read-only, model.
2	JList(ListModel dataModel) Constructs a JList that displays elements from the specified, non-null, model.
3	JList(Object[] listData) Constructs a JList that displays the elements in the specified array.
4	JList(Vector<?> listData) Constructs a JList that displays the elements in the specified vector.

```

import javax.swing.*;
public class ListExample { ListExample(){
    JFrame f= new JFrame();
    JList l1 = new JList();
    l1.addElement("Item1");
    l1.addElement("Item2");
    l1.addElement("Item3");
    l1.addElement("Item4");
    list.setBounds(100,100, 75,75);
    f.add(list);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
public static void main(String args[])
new ListExample();
}

```



JScrollPane:

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

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

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

```

import javax.swing.*;
import
java.awt.event.*;
import java.awt.*;
public class ScrollPaneDemo extends
JApplet { public void init()      {
    Container cp = getContentPane();

    JPanel jp = new JPanel( ) ;
    jp.setLayout( new GridLayout( 20, 20 ) ) ;

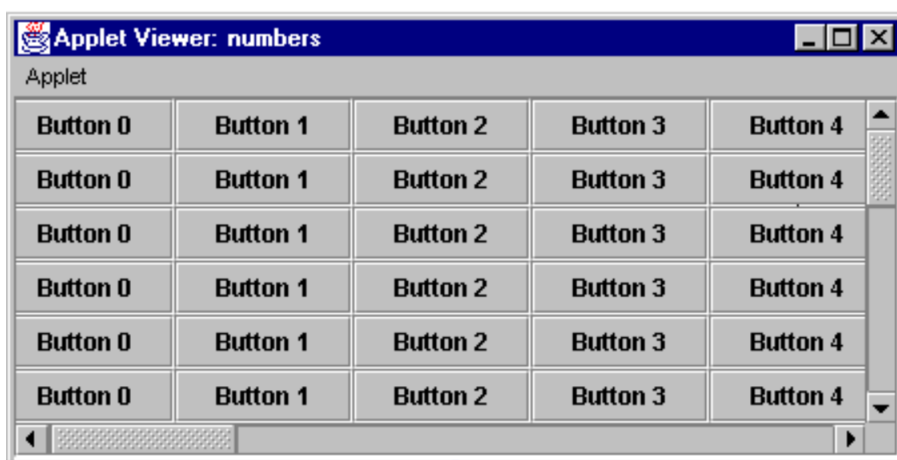
    for(int i = 0 ; i < 20 ; i++)
        for( int j = 0 ; j < 20 ; j + + )
            jp.add(new JButton("Button " +
                                j));

    int v = ScrollPaneConstants. VERTICAL_SCROLLBAR_AS_NEEDED ;
    int h = ScrollPaneConstants. HORIZONTAL_SCROLLBAR_AS_NEEDED ;

    JScrollPane js = new JScrollPane( jp, v,
    h ) ; cp.add(js, BorderLayout.CENTER) ;

```

Output:



Java JSplitPane:

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

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

Constructors

Constructor	Description
JSplitPane()	It creates a new JsplittedPane configured to arrange the child components side-by-side horizontally, using two buttons for the components.
JSplitPane(int newOrientation)	It creates a new JsplittedPane with the specified orientation.

JSplitPane(int newOrientation, boolean newContinuousLayout)	It creates a new JsplitPane with the specified orientation and redrawing style.
JSplitPane(int newOrientation, boolean newContinuousLayout Component newLeftComponent, Component newRightComponent)	It creates a new JsplitPane with the specified orientation and redrawing style, and with the specified components.
JSplitPane(int newOrientation, Component newLeftComponent, Component newRightComponent)	It creates a new JsplitPane with the specified orientation and the specified components.

import

```
java.awt.BorderLayout;
```

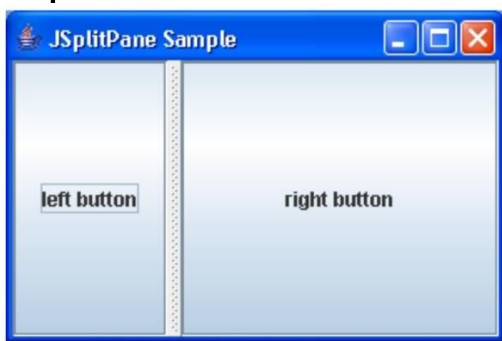
```
import javax.swing.JFrame;
```

import

```
javax.swing.JSplitPane;
```

```
public class SwingSplitSample {
    public static void main(String args[]) {
        JFrame frame = new JFrame("JSplitPane Sample");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JSplitPane splitPane = new JSplitPane();
        splitPane.setOrientation(JSplitPane.HORIZONTAL_SPLIT);
        frame.getContentPane().add(splitPane, BorderLayout.CENTER);
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

Output:



Java JTabbedPane

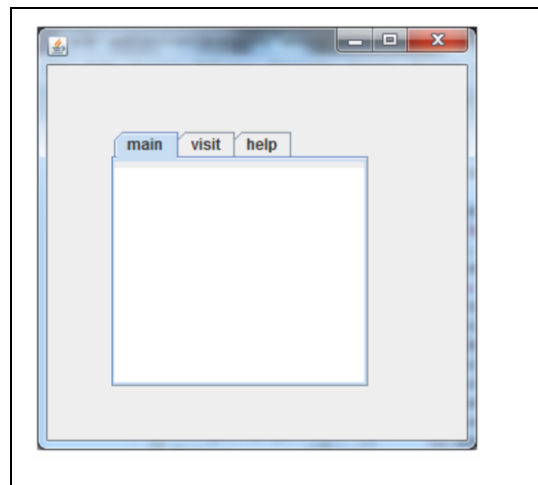
The JTabbedPane class is used to switch between a group of components by clicking on a tab with a given title or icon. It inherits JComponent class.

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


```

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

```



JTree

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

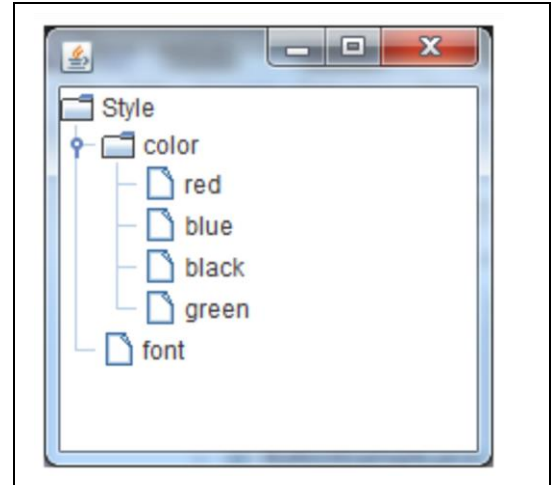
Commonly used Constructors:

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

```

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

```



JTable

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

Commonly used Constructors:

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

JDialog:

The JDialog control represents a top-level window with a border and a title used to take some form of input from the user. It inherits the Dialog class. Unlike JFrame, it doesn't have maximize and minimize buttons.

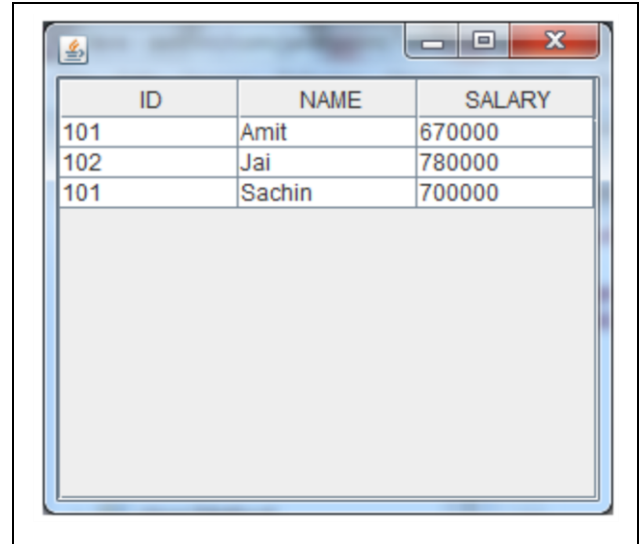
Commonly used Constructors:

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

```

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

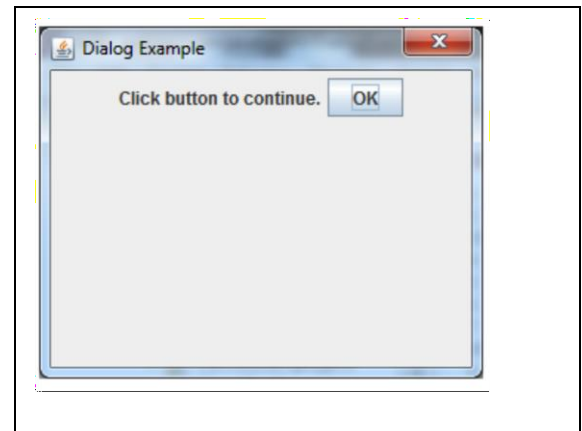
```



```

import javax.swing.*;
import java.awt.*;
import
java.awt.event.*;
public class
DialogExample { private
static JDialog d;
DialogExample() {
JFrame f= new JFrame();
d = new JDialog(f , "Dialog Example",
true);
public void actionPerformed( ActionEvent e
)
{ DialogExample.d.setVisible(false);
}
});
d.add( new JLabel ("Click button to continue."));
d.add(b); d.setSize(300,300);
d.setVisible(true);
}
public static void main(String args[])
}

```



Java LayoutManagers

The LayoutManagers are used to arrange components in a particular manner. LayoutManager is an interface that is implemented by all the classes of layout managers. There are following classes that represents the layout managers:

1. java.awt.BorderLayout
2. java.awt.FlowLayout
3. java.awt.GridLayout
4. java.awt.CardLayout
5. java.awt.GridBagLayout
6. javax.swing.BoxLayout
7. javax.swing.GroupLayout
8. javax.swing.ScrollPaneLayout

9. javax.swing.SpringLayout etc.

Java BorderLayout

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

1. **public static final int NORTH**
2. **public static final int SOUTH**
3. **public static final int EAST**
4. **public static final int WEST**
5. **public static final int CENTER**

Constructors of BorderLayout class:

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

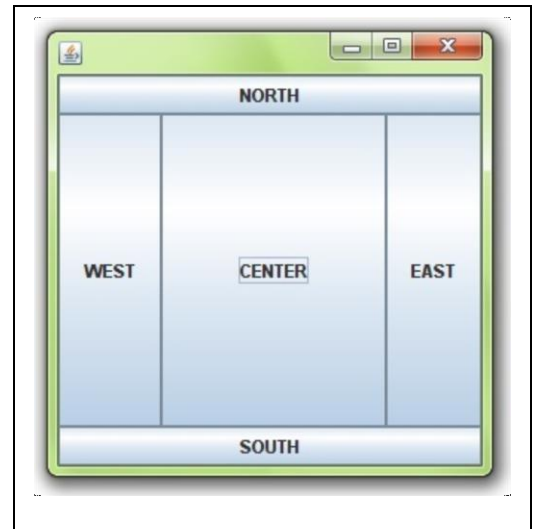
```
import java.awt.*;
import javax.swing.*;

public class Border
{ JFrame f;
  Border(){
    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 Border();
  }
}
```



Java GridLayout

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

Constructors of GridLayout class

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

Example of GridLayout class

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

public class
MyGridLayout{ JFrame f;
MyGridLayout(){
    f=new
    JFrame();

    JButton        b1=new
    JButton("1");   JButton
    b2=new          JButton("2");
    JButton        b3=new
    JButton("3");   JButton
    b4=new          JButton("4");
    JButton        b5=new
    JButton("5");   JButton
    b6=new          JButton("6");
    JButton        b7=new
    JButton("7");   JButton
    b8=new          JButton("8");
    JButton        b9=new
    JButton("9");

    f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);
    f.add(b6);f.add(b7);f.add(b8);f.add(b9);

    f.setLayout(new GridLayout(3,3));
    //setting grid layout of 3 rows and 3 columns

    f.setSize(300,30
0);
```

